Department of Computer Science

Technical Report

Undergraduate Dissertation: A Scripted Sample-Based Music System for Game Environments using .NET

Laurence Parry

Technical Report 2004-14 **May 2004**

ISSN 1740-9497

Contact Address:

Department of Computer Science
University of Bath
Bath, BA2 7AY
United Kingdom

URL: http://www.cs.bath.ac.uk

ISSN 1740-9497

A Scripted Sample-Based Music System for Game Environments using .NET

Laurence Parry

BSc in Computer Science

University of Bath

May 2004

A Scripted Sample-Based Music System for Game Environments using .NET

Submitted by **Laurence O M Parry**

Copyright

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Batchelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Signed: …….. (Laurence Parry)

Abstract

Computer music is an area which has shown much development in recent years. Improvements in consumer sound hardware have allowed computer games to include complex music systems controlled by scripts. We study an example of these, including reverse-engineering of the file format and analysis of the scripting language, for use in creating tools to edit and play this music by third parties. Implementation of a component-based music file editor and player is attempted in .NET using an LALR parser compiler and an interpretation engine based on the Visitor pattern. We provide a specification of the file format and scripting language with the above tools, and conclude that it is possible to develop complex scripted music systems with relative ease using .NET.

Acknowledgements

I would like to thank my project supervisor John ffitch for his valuable support and advice, both humorous and otherwise. I would also thank those members of the *Creatures* community that have assisted in this project, notably MNB, Clucky, vadim, edash and the regulars of Sine and JRC. The *Compiler Tools for C#* provided by Dr M Crowe of the University of Paisley were vital to this project, as were the VBCommenter and NDoc projects.

Contents

List of Figures

List of Tables

Introduction

This project examines a specific example of the use of scripted music in computer games, and follows the development of tools to support its creation and playback. Our objectives were to identify and document the music format (described in chapter 3), decide who the users of this music were and what tools might be useful to them (chapter 4), and then design, implement and test these tools (chapters 5, 6 and 7). The conclusions gathered from this work are presented in chapter 8.

The *Creatures* series (Grand 1997) is a curious mix of game and artificial life simulation. The first version of the game used several short sampled sequences to provide in-game background music. Later games have used a far more sophisticated means of music generation based around a set of short recorded sound samples – each sample is a series of sound data points, typically either a single note or a chord from one instrument – combined with a script providing instructions to the music engine as to which samples to play, and when to play them. (Cyberlife 2000)

The script file is a set of scripts, one for each track. Tracks are a logical division of music – tracks are either associated with an in-game area or keyed to some specific event (for example, the death of one of the creatures). The samples played in each track can be affected by the situation; for example, if an evil creature is around then harsher sounds are used. This is implemented by providing externally modifiable script variables (e.g. "Threat").

MNG[1] (or "munge") files are used to store the game music. The name is apt, as the file format mixes together both the samples used to create the music and the scripts directing their use. Preliminary work authorised by Creatures Labs as part of their permission for this project had partially identified the layout of this file through reverse engineering techniques – the original music was created several years ago and there was no documentation available for the file or scripting formats. The first objective of this project was to document these formats.

There are many thousands of *Creatures* users around the world, and many of them have created their own additions to the game; *Creatures* has always supported third-party additions, and latter versions of the game are highly customizable. These additions include *metarooms*; areas of the game environment that have been added to the base system, or which replace portions of it altogether. The designers of these additions wished to add appropriate music to their metarooms, but were unable to do so because there was no tool to create new music files. Creation of such a tool was the next (and primary) objective of this project.

This editing tool was useful, but hard to use alone as it did not allow users to preview the music that they were creating, requiring them to load it into the game. To improve this situation, another identified objective was to make a sound engine and accessory capable of playing the music specified by the script. This required parsing and interpreting the script, sound mixing, application of effects and output of the resulting music using DirectSound (Microsoft 2003b), and was ultimately successful despite setbacks. The combination of the

[1] Note that MNG is also the acronym for **Multiple-image Network Graphics**, a format for animated image files based on the PNG image format. When the *Creatures* music format was created, the MNG acronym was PNF.

two tools was successfully used by several target users to add music to their metaroom projects.

Finally, as scripting does not tend to be particularly intuitive for non-programmers (including most musicians), allowing the users to create music without knowledge of the underlying scripting language was considered useful. This required some form of interface that was easy to learn yet exposed all features available in the language. The final aim of this project was to design and implement such an interface for the editing tool. This aim was not fully realized; however the information collected and components generated while creating the previous tools make its completion within a few months a possibility.

Literature Review

Overview

There were many factors to be considered before undertaking this project. The purpose of this review is to identify these factors, and to understand their relevance to the tasks that will be performed. It will also explain the context of this project in relation to other works, especially in the academic world. In particular, it will show how the union of computers and music has developed over the years, and how this project makes use of the research and tools that have been created to fulfil the needs of computer music.

We will begin by looking at the history of computers and sound, and how it has developed over time into the field of computer music. This leads us into the more recent topic of music in computer games, as well as to the subject of aleatoric music. A large amount of research has been done towards providing effects to modify sounds and music and we shall give a few examples of these.

We shall then proceed to consider the basics of scripting languages, both their purpose and use, and the compilation techniques that may be useful to interpret them for this project. Finally, we will give some thought to the other technologies which might be suitable for implementing this project – in particular, the choice of programming language – and close with a summary of the points raised.

Computers Sound and Computer Music

The field of computer music dates back to the late 1950s when Max Mathews developed the pioneering *MUSIC4* at Bell Laboratories, one of the first sound synthesis programs (Burns 1997). He and fellow researchers would go on to produce further versions, including a portable *MUSIC5* written in FORTRAN (Mathews 1969). More importantly, they shared their knowledge with fellow researchers in Princeton and Stanford. However, computers of the time could not compete with analogue electric synthesizers in terms of audio quality, and it was not until the 1970s that digital generation of sounds became popular. Efforts proceeded both in the area of sound generation and the playback of digitally stored sounds, which was previously unfeasible owing to storage constraints.

The invention of *MIDI* (MIDI 2003) in 1983 provided a standard for sequenced computer-based music. Two years later Barry Vercoe invented the *Csound* system (Csound 1999), a sound synthesis and scoring system in the spirit of the then long-lived *MUSIC* series, which he had developed through the late 60s and 70s. More recently, personal general-purpose

computers have become sufficiently powerful that complex instrument synthesis and effects processing can be performed in real time on them without dedicated DSP hardware. This has allowed new applications in many areas – most noticeably in computer games.

An area of research that it is useful to contrast this project against is algorithmic composition, as described in (Maurer 1999). This has always been a popular topic in research circles, and it is still an active area of research. Many systems have been developed in an attempt to generate "good" music through rules-based or stochastic methods, or more recently through the use of genetic algorithms. This project does not include such techniques – it relies on the composer to generate a script, and so a human performs the act of composition; the execution is left to the computer. However, certain random and computer-controlled elements remain, notably the use of in-game variables to influence timings and instruments used.

Nowadays there are two main extremes in the computer music world – those who seek to generate music from samples of real sound, and those who prefer to generate music from entirely artificial sounds. Those in the commercial world tend to use the former as they have greater control over the output and, while many of those in the research community prefer the latter. It is our belief that a mixture of these approaches is useful, and indeed many contemporary music generation schemes fall somewhere in-between; in particular, the use of recordings of computer-generated sounds is common where on-the-fly generation of those sounds would be computationally infeasible, and the application of computer effects to natural (or at least non-computer-generated) sounds is a large portion of this project.

Music in Computer Games

Music was often a selling point for personal computers of the early 80s, most notably the Commodore Amiga and Atari ST, the latter of which was advertised under the tagline "This Computer Was Made For Music" (Morton 1990). In stark contrast, the IBM personal computer was not supplied with any built-in sound capability – it was designed solely as a business machine. Those who used it for other purposes were forced to wait for additional hardware to become available.

The first add-on sound card for the PC to achieve mass-market popularity was the *Adlib*. This was later overcome by the *SoundBlaster* system, which survives today in the *Audigy* family of cards. These add-on cards allowed stereo sound and enveloped instruments, and became widely used both to generate music and in-game sound effects. Most used frequency modulation (FM) techniques discovered by Chowning (1973) and first implemented in Yamaha synthesizers.

As with FM, wavetable synthesis was first used in electronic synthesizers. Wavetable synthesis works by taking several digital samples of an instrument and playing these when directed (typically by a MIDI score). This technique is now in general use, having surpassed the FM technologies used by earlier sound cards, and the use of FM to generate custom sounds has been replaced by the practice of using recorded samples.

However, computer music is not all about what techniques can be used to generate the sounds. Software is also required to control when and how sounds are to be played to form music. This can be decided by some external body of code – i.e. the main game – and so make use of its knowledge of the status of the game and any events that may occur. This

technique is known as adaptive audio (Clark 2001, Whitmore 2003). Such systems significantly aid the generation and use of music to match the situation; however, it is generally recognized that such technical aids still require experienced composers to produce desirable results. Most musicians in the gaming industry still do not consider algorithmic composition sufficiently advanced to produce the required quality of music on its own.

LucasArts was regarded by many as being one of the leaders in this area during the early 90s with their *iMuse* system. Developed by Land and McConnell (1991), *iMuse* was originally designed to handle MIDI streams, and later extended to apply to sampled sounds. Naturally, this has a direct relevance to our project. The main difference is that *iMuse* was designed to control the output of pre-prepared MIDI or digital scores, whereas the *Creatures* music system is designed to generate music from samples; its script allows for variations in the timing of samples, or whether they should be played at all. The former method makes it easier to generate complex music, but the latter allows for a more aleatoric approach. *iMuse* was certainly very successful, and is a good example of how event-based music can add significantly to the perceived quality of a game.

This field moves relatively quickly – in the space of just a few years we have come from monotonic beeps to quality rivalling professional sound systems; indeed, many composers now base their entire work around computers. This progress has been driven by the demands of consumers for high-fidelity sound, increase in the numbers of these consumers with the rise of the PC gaming industry and rapid advances in the technology available to implement such features.

Aleatoric Music

Aleatoric music is variously defined as “Chance music in which the performers are free to perform their own material and/or their own manner of presentation,” and “Composition depending upon chance, random accident, or highly improvisational execution, typically hoping to attain freedom from the past, from academic formulas, and the limitations placed on imagination by the conscious mind.” (Delahunt 2003)

It may seem difficult to reconcile the idea of chance with computers, which are by their nature deterministic machines. However, many algorithms have been developed to generate pseudo-random numbers, and the speed at which computers can perform this has provided many applications. In the last century, much work was done in this area by the American composer John Cage (of *4’39”* fame), who made use of computer programs in concert with Andrew Culver (2001) to generate note timings, pitches and even whole pieces. Still, this work was generally done as a single piece rather than being controlled through variables that may change as the result of external actions.

The music in *Creatures* does not aim to be totally aleatoric – rather it incorporates aleatoric elements, such as partially random delays between instruments. This allows an element of irregularity to enter the composition while affording the composer a great deal of control – as mentioned in the Cyberlife article on the music (2000), “you hear a melody that was never intended as the brain makes sense of the different sounds generated.”

Sound Effects

Much of the variation and liveliness in the music for the *Creatures* games is produced by the use of various sound effects. Sound effects are a specific application of digital filters and digital signal processing, which is a whole topic in itself (Smith 1999). In essence, they can be regarded as the convolution of one waveform with another – for example, an echoed version of a given sample can be produced by taking the sound of a sudden balloon burst and the subsequent echoes, and applying it to the sample waveform with convolution. However, this method is typically used only for complex reverberations and other effects that are impossible to apply otherwise, as they are computationally expensive. More typically, a filter is some combination of the input to and/or output from the filter over a number of data points in a sample.

Other examples of sound effects are *chorusing*, which may be implemented by playing the same sound at slightly different intervals (and slightly out of tune); *flanging*, which causes the sound to appear to “speed up” and “slow down” over a short period of time; *compression*, that restricts the range of amplitudes (causing quiet sounds to become louder and loud sounds to become quieter) and *wave-shaping*, which can alter the timbre of a sound by applying a function to the amplitude that has a particular shape.

Effects must be used judiciously, lest they overwhelm the original sound (and in the case of computationally expensive filters such as reverberation, the computer on which they are running). Care must also be taken to ensure that i (where the sound level suddenly changes from one level to another) and/or *clipping* (where the sound produced exceeds the maximum level expressible) do not occur; the latter may be ameliorated by the appropriate use of *gain reduction*, a technique whereby the signal level is reduced to fall within the valid range.

Scripting

Both samples and effects require orchestration. In the MNG music system these are specified by the composer in a custom scripting language. There are many definitions of exactly what a scripting language is. John Osterhout (author of Tcl) proposed the following dichotomy:

> *System programming languages (or "applications languages") are strongly typed, allow arbitrarily complex data structures, and programs in them are compiled, and are meant to operate largely independently of other programs. Prototypical system programming languages are C and Modula-2.*
> *By contrast, scripting languages (or "glue languages") are weakly typed or untyped, have little or no provision for complex data structures, and programs in them ("scripts") are interpreted. Scripts need to interact either with other programs (often as glue) or with a set of functions provided by the interpreter, as with the file system functions provided in a UNIX shell and with Tcl's GUI functions.*

This definition is debatable, particularly as the line between scripting and system languages has narrowed. For example, PHP would seem to be a scripting language, yet there exist accelerators for PHP that effectively turn it into a compiled language. (Zend 2003)

Regardless of the definition, scripting languages have arisen because there is a need for high-level, easy-to-learn and readily modifiable computer languages. In the case of this project, the advantages of using a scripting language to control the music are twofold; it allows rapid changes to the music (possibly even while the script is running) and does not force the composer to learn a programming language in order to do their work. Given the simplicity of the scripts, efficiency is not a significant concern. However, we do need to be able to read the scripts, and this requires the use of technologies used for compilation.

Compilation Techniques and Technologies

At a first glance, compilation does not seem to have much to do with the generation of computer music. However, compilation is often viewed as a translation from one language to another. This process is similar to that of music, which is translated from one form – notes on a sheet of music – into another, the performed piece, by the performer. In this case, the performer has taken on the role of the translator.

In this project it was necessary to translate from the written music script to a representation that may be used by the program, such as an abstract syntax tree. This involves the processes of lexing (separating and identifying unique tokens in the input stream) and parsing (putting these tokens together to form meaningful statements).

There are many ad-hoc methods available for performing this task, but the standard tools for creating lexers and parsers are *lex* (Lesk 1975) and *yacc* (Johnson 1975). These tools – known respectively as lexical analyzer generators and parser generators – allow the vast majority of computer languages to be processed[2]. This is done by preparing (for *lex*) a token recognition mapping and (for *yacc*) a file indicating the rules for matching patterns of tokens and the actions to be performed when a match is found – the grammar of the language.

These particular tools are proprietary, developed by AT&T in the 70s. Improved versions called *flex* and *bison* have been created and distributed under free open source licenses by the GNU project (GNU 2003a, 2003b). Use of these tools allows the programmer to avoid the complicated task of recognizing syntax and to concentrate on semantics.

All the tools mentioned create C source code, and so are best suited for programs written in C or C++. However, this should not curtail our choice of language, as there are many alternatives claiming compatibility or equivalent function for other languages. In particular, the JFlex lexical analyzer (Klein 2003) and the CUP parser generator (Ananian et. al. 1999) are available for Java, and similar tools have been created for C#; due to the language-agnostic nature of the .NET platform, this means that all the .NET languages have access to them (Crowe 2004).

Implementation Technologies

The *Creatures* games run on Microsoft Windows, and so this was the initial target platform for this project. There are of course many languages available to implement this project in. Indeed, some music-specific languages were considered (Thompson 2003), but owing to the large amount of non-musical content in this project, a general-purpose language was deemed more suitable.

[2] Specifically, those for which it is possible to construct an LALR context-free grammar

The original music system was created using C++, and this was a clear option, featuring high performance and a wide user base. However, since C++ is a relatively low-level language, development can take a relatively long amount of time, and as time was limited a higher-level approach seemed more suitable.

Visual Basic was considered, as it provides rapid application development and is also widely used. Unfortunately it lacks a good freely available implementation of lexing/parsing tools, without which we would have had to generate our own parser logic, taking time away from the core objectives. It also lacks key language concepts such as inheritance. (McKinney 1997)

Another candidate language was Java. This has a strong class library system, an object-oriented approach and good user-interface support. However, it is weak in terms of sound functions – as it must be general across several platforms, it is unable to use the strengths of each to full effect, and so the sound API (Sun 2003) is relatively basic, offering no support for advanced effects.

This led us to look at the .NET platform, and in particular Visual Basic.NET. This incarnation of Visual Basic includes proper exception handling and a vast library of classes - the .NET Framework (Microsoft 2003a). Running on this framework provides similar garbage collection and security benefits to Java. Moreover, use of the .NET platform would not preclude us from using the native DirectSound API (Microsoft 2003b), a particularly useful feature as the original implementation used DirectSound to manage both sound output and effects. Using DirectSound might allow the use of chorus, compression, distortion, echo, flanging, gargling, equalization and environmental reverberation without significant extra work. In addition, the lexical analysis and parser generator previously mentioned (Crowe 2004) would be available.

Naturally for a project intended for use by others, portability is a concern, as currently the main implementation of the platform is only for Windows. However, the .NET platform is (theoretically) an open standard, and there are at least two alternative free implementations in progress – dotGNU Portable.NET (DotGnu 2003) and Mono (Mono 2003). Microsoft have also released a “Shared Source” version of .NET called Rotor (Microsoft 2002) working on Windows, FreeBSD and (recently) Mac OS X 10.2. This was deemed sufficient as the primary objective of this project was a system running on Windows. Further details relating to portability concerns are given in the conclusion (see 8.5.5).

DirectX also offers a music API – DirectMusic (Hays 1998, Yackley 1999). However, we believed there to be several good reasons not to use DirectMusic to handle the music. One reason was the lack of a “managed” version of the DirectMusic API for .NET, as there is for DirectSound. Writing an interoperability layer between .NET and DirectMusic would be a significant task. Another was that DirectMusic is still heavily tied to the MIDI approach of sequenced music (albeit with the option of sampled sounds), while our system was based around a potentially more generative, scripted approach.

Moreover, using DirectMusic would further tie the project to proprietary technologies for which there is no open-source implementation available. DirectSound is proprietary, but with appropriate design it could be replaced by another mechanism for output and effects, just as the implementation of the .NET platform itself could be changed, without materially affecting the project. It is rather less likely that a third party will produce a DirectMusic-compatible component, and so we deemed it unwise to use DirectMusic in this project.

Summary

We began by considering the general development of computer sound and computer music from its origins in the 1950s to today. We have seen that over time, new technologies and techniques became available for the generation of sounds and music, including frequency modulation, wavetable synthesis, algorithmic composition and the MIDI standard for sequenced music. We then continued to a discussion of the various types of hardware device used for the playback of music, and how these have changed over the years in response to user demand.

Adaptive audio was introduced as an answer to the need for event-driven software control of music, and a discussion of the *iMuse* system, and its relevance to this project was given. A short diversion into aleatoric music was made, and its historical use by Cage briefly discussed. We then proceeded to outline the concept of sound effects, and suggested a few examples and how they might be implemented.

We then gave a definition of scripting languages, and offered several reasons why use of a scripting language was an appropriate choice for the composition of music. The evolution of lexical analysis and parser generation was shown, and a variety of implementations of these were considered. Finally, we investigated several languages and technologies and gave our reasons for using a combination of Visual Basic.NET, DirectSound and the C# compiler tools for the implementation phase of this project.

Preliminary Work

Our lack of knowledge regarding the music format made it clear that some preliminary work above and beyond the literature survey was required before embarking on any design. This would both serve as a solid base for this project and allow others to create their own works using MNG files. We regard the information provided in this chapter and in Appendix A sufficient to create programs similar to ours in any language.

Identification of the MNG File Format

Little was known about MNG files at the start of this project. The only information available to us was a short article detailing some of the features of the Creatures sound engine (Cyberlife 2000), as the persons involved in making the music engine had left the company and no written documentation was available.

From this article we knew that:

- The music was intended largely as background “mood” music
- There were individual soundtracks for the separate areas of the game, as well as for a few key events, such as the death or birth of a creature
- Each soundtrack was composed of several “players” and a script

- The music used sampled sound rather than MIDI – this was already clear from looking at the size of the file
- The music incorporated "feedback loops"
- Variables such as "threat" and "mood" affected the music:

> *"Behind the scenes, scripts control the music engine and set the volume, panning and interval between notes as the mood and threat changes. When a Norn is lonely or hungry, the mood score is low. Alternatively, when a Norn has just eaten and is happily playing with a toy or with friends, the mood score is high. As the threat level increases so does the volume and you get a sense that something is about to happen."*

Given the size of the file, the amount of music, and the other files used for sound effects, we suspected that the music was stored as samples for each instrument in WAV format (IBM and Microsoft 1991), the standard for Windows recorded audio. However, on scanning with a hex editor, we failed to find any telltale WAV file signatures ("RIFF" or "WAVE"). Instead, a regular structure appearing to be an index of positions and lengths was found at the beginning of the file, starting with a number which we took to be the number of samples in the file.

Our original hypothesis was eventually confirmed by loading the file into a sound editor (Cool Edit) as if it were a raw PCM encoding. Several distinct areas after the file were immediately identified, and all but one of these were recognizable as sounds; the first was later found not to be a sample at all. We were also able to identify the sample rate, quantization level and number of channels (22050 Hz 16-bit mono) by comparing with original sounds, which proved useful later.

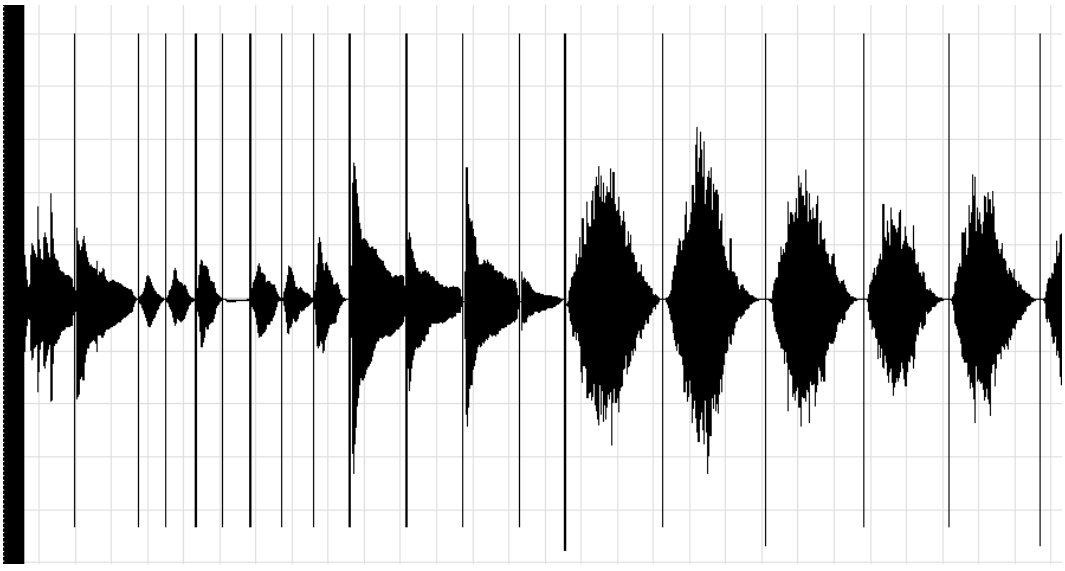


Figure 0-1: Samples of sound are clearly visible within the MNG file

Having confirmed that sound was stored in a PCM encoding, we attempted to classify the rest of the file's contents. Each sample began with a short burst of static, which (from its regular

layout) indicated header information of some sort. The breakthrough occurred when examining the numerical values of one of these bursts of static. One of the word values contained in the file was 0x2256, corresponding to the little-endian hexadecimal value of 22050, the previously identified sample rate. This corresponded to a section of the WAV header, and other portions of the file corresponded with valid values for a WAV header.

The format was finally identified as a WAV file, with the first 16 bytes truncated. Why the WAV was truncated was unknown; it may simply have been for ease of implementation, or it may have been a security measure to discourage tampering/theft of the samples or unauthorised creation of new files.

Having identified the physical layout of the file, we proceeded to decoding the script. It was clear that this was likely to lie within the first "chunk" of the MNG file, but the chunk appeared to be formed of semi-random data. No clue as to its format was available in the written materials available to us – they had not discussed implementation details.

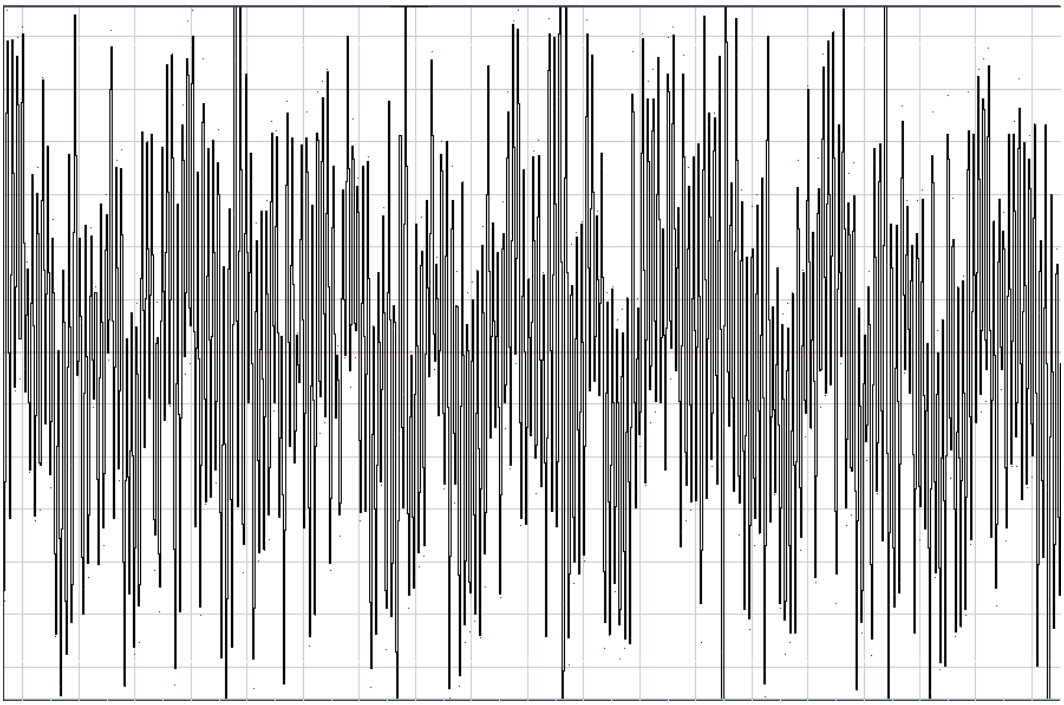


Figure 0-2: A rough pattern is present in the scrambled script

At this point we decided to look at the other files available to us. In particular, MAP files containing debugging information (most usefully the names and offsets of all class methods and functions) had been provided with the game executables for use by the game's crash reporting function, and so we examined those. The *LoadScrambled* method of the *MusicManager* class was of immediate interest, as was the *Scramble* function.

Use was made of a program called the *Interactive DisAssember* (IDA 2004) to disassemble the code involved. This tool was chosen for its ability to present the disassembled code in an easily navigable format. The *Scramble* procedure was relatively short (as documented in Appendix D) and a VB.NET equivalent is shown in Appendix A – it was found to perform a

reversible transformation using the bitwise XOR function. This is the same function used in most *one-time pads* (see Wickepedia 2004), and is generally considered to be unbreakable if the associated "pad" of bytes to be XORed against is random and of sufficient size. However, since the game needs to descramble the code, it has to be provided by the game. In fact, the pad used was merely an incrementing byte counter, providing a very simple *stream cipher*.

If the MAP files had not been available the task would have become considerably more difficult, as knowing which procedure to consider would have required tracing the flow of execution and looking for references to the MNG files. It might even have been possible to attack the cipher using frequency counts alone; space characters form a high proportion of the output, and this could have been predicted without seeing the resulting script. Since the generated XOR pad was incremental, and most ASCII characters are within a certain range, there was a rough pattern to the scrambled output from which the encryption method and pad might have been deduced; it would, however, have been a complex task.

After accounting for every byte in the file, it was still not at all clear where the identifiers for the music were stored. Finally we realised that the samples were named implicitly – the first sample referred to in the script in a voice's Wave declaration was the first sample stored in the file, the second was placed after that, and so on. We believe that the original *Creatures* music files were constructed from a collection of WAV files, and so easy identification of the samples was not an issue.

Overview of the MNG Scripting Language

At this point we shall present an overview of the scripting language, based on our analysis of the scripts in the game music files. The scripts contain instances of inconsistent formatting, leading us to believe that the game scripts were created manually – it also would not make much sense for there to be a tool to create them, as the development of music is a solitary task, and it would be cheaper and quicker to train the few intended users in the scripting language than to write such a program.

Many of the script commands were readily understandable by their names – others required experimentation to identify their function. Further information was provided by the CAOS scripting language guide (Cyberlife 2004) – CAOS is used in Creatures to control objects and respond to events, and it contains several commands allowing CAOS scripts to control the music, including those to changing the currently-playing track, set the threat value of an object, and fade out the music.

Tracks

The basic unit of music is a Track, which in the game is associated with either a specific event - for example, the death of a creature - or an area, like the Volcano track. Typically only one track plays at once; switching between tracks is accomplished by fading in and out, adhering to the FadeIn and FadeOut track parameters. Each Track has one or more Layers, which are played simultaneously.
Comments are indicated by a double slash (//) and may be placed within Track, Effect, Voice or Update declarations, as well as at the top level.

```
Track(UpperTemple)
    {
    FadeIn(5)
    FadeOut(5)

    LoopLayer(Chord)
        {
        …
        }

    AleotoricLayer(StickMelody)
        {
        …
        }
    }
```

Variables

MNG scripts have a concept of local variables, which reside within these layers. Variables must be declared before use, with the name and an initial value. The variables are floating point values associated with names. Some variables are special – Pan and Volume because they affect the samples to be played, and Interval because it affects the length of the track.

```
AleotoricLayer(Pad)
    {
    Variable(temp,4.0)
    ...
```

Layers

Layers are the "instruments" of a track, in that they either play one sample repeatedly (in the case of LoopLayers) or one or more samples, enclosed within Voices (AleotoricLayers).

LoopLayers

A LoopLayer consists of a single Wave and an Update block. The Wave is played constantly and repeatedly. The Update is called at regular intervals and typically causes some change in the presentation of the samples (for example, it may pan the output from side to side, or alter the volume).

```
    LoopLayer(HighBreath)
        {
        Variable(counter,0.0)
        Variable(temp,0.0)
        Update
            {
            // Gradually, pan around at a random rate
            temp = Random(0.0, 0.1)
            counter = Add(counter, temp)
            Pan = CosineWave(counter, 30)
            // Scale the volume according to mood
            Volume = Multiply(Mood,0.4)
            Volume = Add(Volume,0.6)
            }
        UpdateRate(0.1)
```

```
Wave(HighBreathG)
}
```

AleotoricLayers and Voices

An AleotoricLayer consists of one or more Voices to be played sequentially. Effects and Volume may be specified for the layer. The Interval of a layer specifies how long it is before the next Voice of an AleotoricLayer is to be played – it is possible to change this within the Voice.

Voices are individual Waves with optional Conditions and Intervals. Conditions are used to decide whether or not the Wave should be played – the value of the specified variable must be between the two specified values. Intervals allow the script to specify how long to wait before the next sample.

```
AleotoricLayer(BendyEcho)
    {
    Volume(0.4)
    Effect(PingPong160)
    Interval(4)
    Voice
        {
        Condition(Mood,0.2,0.6)
        Wave(Bnd0)
        Interval ( Random( 4.0, 9.4) )
        }
    Voice
        {
        Condition(Mood,0.4,1.0)
        Wave(Bnd1)
        Interval ( Random( 4.0, 9.4) )
        }
    }
```

Update Blocks

Both the LoopLayer and AleotoricLayer structures may have one Update block. This block consists of assignments to variables (which may be special variables such as Volume or Pan) that are carried out each time an update is called, and also when beginning to play a layer. The time period for updates is set by the UpdateRate or BeatSynch statement in LoopLayers, or each time the last Voice is considered for AleotoricLayers. The Update blocks may also be placed within Voice blocks, in which case the update takes effect after the voice's Wave has been played.

```
AleotoricLayer(Pad)
    {
    // The track sparsely plays pads ranging from the gentle (drm)
    // for low threat, with harsher (vce) for heigher threats
    // Volume increases with mood and threat,
    // The interval is decreased with threat
    Volume(0.4)
    Variable(temp,0.0)
    Update
```

```
        {
        // Volume = 0.5 + 0.25 * (Mood + Threat)
        temp = Multiply(Mood, 0.25)
        Volume = Add(0.5,temp)
        temp = Multiply(Threat, 0.25)
        Volume = Add(Volume,temp)
        Interval = Random ( 4.0, 6.0 )
        temp = Multiply( Threat, 2.0)
        Interval = Subtract( Interval, temp)
        }
    …
```

Intervals

Intervals represent a pause in the output of a layer, either between Voices if specified for a particular voice or between iterations of a layer if in the main body. Processing of a layer does not continue until a pause for the length of the interval has taken place. The expression is evaluated anew each time.

Beats, BeatLength and BeatSynch

Beats are an alternative method of specifying intervals between periods of music. Instead of directly specifying a length in seconds, the BeatLength is specified in the Track, and a BeatSynch is given that measures an Interval in this number of beats. The following specifies an interval of 0.3 * 16 = 4.8 seconds for the Guitar:

```
Track(Underwater)
    {
    BeatLength(0.3)

    AleotoricLayer(Guitar)
        {
        BeatSynch(16.0)
…
```

Effects

The script also specifies Effects, which are preset sequences of setting changes applied to AleotoricLayers (**not** LoopLayers). An Effect has one or more Stages, each of which may make changes to the panning or volume for that layer. After a specified delay, the effect moves onto the next Stage in the sequence.

Effects are applied to the output of a Layer; they essentially take this output and repeat it several times[3]. How many times is defined by the number of Stage declarations in the Effect. As an example, a simple effect might "bounce" the sound from one side to the other, slowly fading the volume at the same time.

Each Stage contains a declaration for the Volume to play the output at, the Pan value (how far to the left or right of centre the sound should be played) and either a Delay or TempoDelay,

[3] The original specification incorrectly identified effects as altering the parameters of **one** output – the results of this misconception are detailed in the sections on Implementation and Testing.

indicating how long to pause before moving on to start the next effect. Values may be expressions or constants. TempoDelays are present in effects intended for layers using the BeatSynch and are measured in beats as defined in the Track currently playing, whilst Delay is measured in seconds.

```
Effect(RandomPad)
    {
    // Produces randomly panned echoes, staggered at close
    // random times
    Stage
        {
        Pan(Random(-1.0,1.0)) Volume(1) Delay(Random(0.25,0.4))
        }
    Stage
        {
        Pan(Random(-1.0,1.0)) Volume(0.92) Delay(Random(0.25,0.4))
        }
    Stage
        {
        Pan(Random(-1.0,1.0)) Volume(0.84) Delay(Random(0.25,0.4))
        }
    }
```

Requirements

Having analysed the MNG file format, we proceeded to consider the potential users of the proposed system and their requirements – as time was limited, it was considered foolish to concentrate on features unwanted by users. However, as often happens the requirements were not clear, and several iterations of prototypes would prove to be necessary before a final list could be made.

The Users

No project can succeed without considering its users. In this case, the prime targets for the tools to be developed were metaroom developers. However, during requirements solicitation two further group of end-users were identified – software developers and general users, as detailed below.

Metaroom Developers

A *metaroom* is the basic unit of third-party world expansion to the *Creatures* and *Docking Station* games. Metarooms are expansions to the living space within the game; high-quality, original metarooms are in great demand by players.

At the time of writing, over twenty metarooms have been created, and several are in development. It is possible for metaroom authors to specify music for the various sections of the rooms in the map editor. However, none of the metarooms were able to incorporate their own music, as no editors were available. The reason for this was that the specification of the MNG music format was unknown, unlike those for other file formats used in the games, which were available from the Creatures Development Network (Gameware 2004).

The initial objective for this project was to identify the format of MNG files and proceed to develop an editor for use by these developers.

Other Creatures Developers

While discussing the project, other developers in the *Creatures* community expressed an interest in developing programs that could read the MNG format, for their own purposes. Some developers intended to make their own MNG handling routines, and for these a detailed file format specification and script decryption algorithm would be sufficient, but others desired a more comprehensive solution.

To this end it was decided that the applications to be developed should be split into components, and source code should be freely-available for non-profit use at the SourceForge online code repository (SourceForge 2003). This also provided a secondary backup for the project source files. The need for proper documentation of the code was also strengthened by the presence of this user group.

General Users

In addition to the separate developers, several *Creatures* users mentioned that they would enjoy listening to the music outside of the game. Indeed, some had separately noticed that the MNG files contained music samples, and extracted them with a sound editor – however, not being programmers they were not aware of the presence of the script, nor would they have been able to play the music if they discovered it. Since we believed that metaroom developers would also find it useful to be able to preview their music outside of the game, creating an accessory that played selected tracks in the background was added as an objective.

Potential users covered a wide range of ages and ability – from seasoned Microsoft programmers to children barely into their teens and inexperienced with anything but games. Of course, all user groups appreciate simplicity, and so an important aim for the player was for it to "just work" – there should be no confusing options to get in the way. However, it still had to be powerful enough for metaroom developers to be able to test their music.

Requirements Solicitation

Requirements solicitation was not especially challenging, as the community was known and readily accessible and we were generally aware of the projects being undertaken. In particular, we were able to pose informal questions to potential users on a regular basis from the start, and could also work with the creator of one of the most advanced "metaroom" projects, C12DS (Creatures 1 to Docking Station), for which a prototype of the editing tools was successfully used.

The main difficulty encountered was in explaining to others the capabilities and limitations of the MNG format and the *Creatures* sound system – some people expected too much from the software, while others thought that (for example) it was restricted to playing a single stream of recorded sound per track, in a similar manner to CD audio.

Throwaway Prototype

It was clear from the beginning of the project that there was insufficient knowledge of the problem to immediately implement an optimal solution. However, users are notorious for not knowing exactly what they need, so rather than just asking them what they thought they would like, we decided to provide them with a basic editor offering the minimum features required to make and modify MNG files (but not the expected levels of robustness or

performance), and invite them to comment on its features and suggest possible improvements.

Rationale

Throw-away prototyping has the well-understood advantages of allowing designers to identify requirements and test assumptions before committing to a final specification or writing any lasting code, and without polluting the final product with non-production-quality code (Brooks 1995). In this case it also had the fortunate side-effects of allowing "power" users to modify MNG files well before the tools provided by this project would otherwise have been available, and allowing us to create custom test cases with relative ease.

Construction of the Prototype

The prototype was constructed in Visual Basic, as the issues cited against its use for the main project were not an issue for the purposes of the prototype. Using a different language also enforced the philosophy of "throw the first one away". The only elements shared between the prototype and latter versions were a few algorithms used for loading MNG files – and, of course, the lessons learnt. The prototype was constructed as the preliminary investigation was underway, and formed a convenient test-bed for theories on the construction of MNG files.

Features of the throw-away prototype included:

- Loading and saving of MNG files
- The ability to add, remove and rename samples
- Manual modification of the music script by loading a script file
- No attempt at parsing the script other than to retrieve sample names
- No attempt at providing a good implementation in terms of robustness, modularity, maintainability or commenting

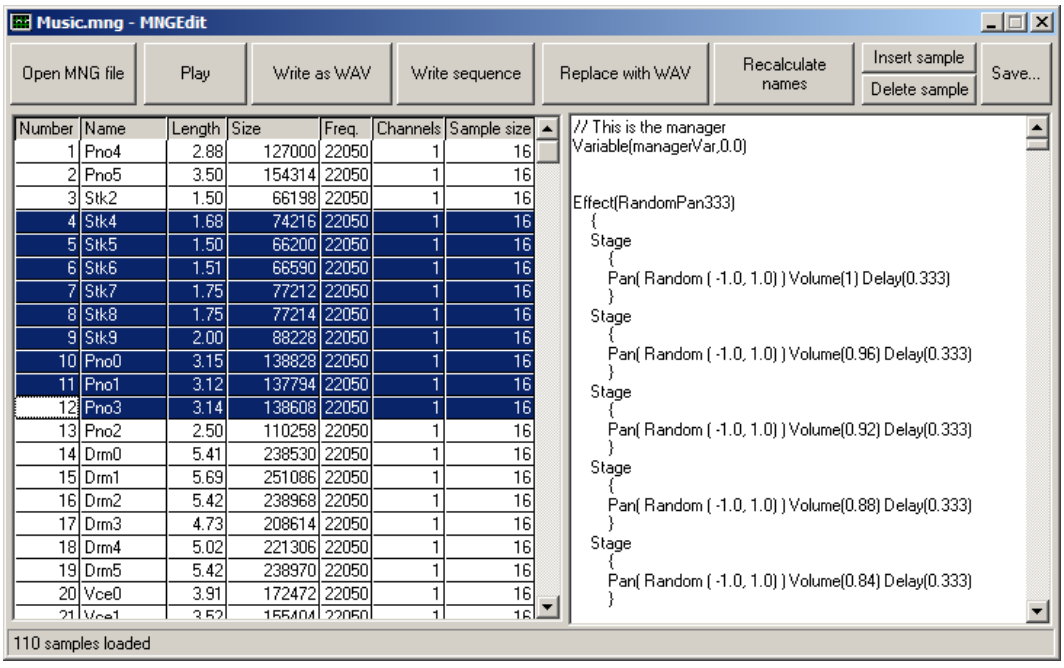


Figure 0-1: The prototype editor was functional, but not generally usable

Questionnaire

Sample users were drawn from the *Creatures* community by announcing the objectives of the project and inviting interested parties to try out the prototype and comment on its features and shortcomings (see Appendix C). There was an known element of bias in this – only users most comfortable with installing unfinished software were likely to respond, and we judged that these were also likely to be the most technically able. For this reason some of the more advanced suggestions were later given a lower priority in preference to usability requirements.

Requirements Analysis

Operating Environment

The tools should be capable of running on any system with support for the .NET runtime and Microsoft's DirectSound libraries. It would be an additional bonus if the editing tools worked without DirectSound, as non-Windows platforms are not likely to support the Managed DirectSound extensions.

Limitations of End Users

The majority of *Creatures* players are children in their early to late teens. While many users of the editing and playing applications will be familiar with scripting languages, possibly through use of the *Creatures* scripting language, CAOS, some may not be particularly able with technology. As a result, while modifications to the game are usually performed by more technically-minded users, some concepts are likely to require careful explanation.

Task Analysis

Users wish to:

- Play music
- Create and edit music files, by
    - Adding and replace samples in music files
    - Editing the script to change how the music is played

Developers wish to:

- Understand how MNG files work
- Develop applications that use the MNG format
    - Open MNG files
    - Access and modify sample data
    - Access and modify the script,
        - Both in raw and tree form

Requirements Specification

1. The simple MNG editor shall
    a. Allow the user to load, save, and create new MNG files

   b. Allow the user to add and remove samples to, rename, and export samples from the MNG file
      i. The samples shall be in WAV format
      ii. Samples may be added and removed by drag and drop
   c. Provide for manual editing of the script file
   d. Allow the user to validate the script to ensure that it is (syntactically) correct
   e. Warn the user if they attempt to save a file which
      i. Does not parse as a syntactically-correct script,
      ii. Refers to a sample or effect in the script that is not present, or
      iii. Contains a sample that is not referred to in the script

2. The MNG player accessory shall
   a. Let the user play linearly through a MNG file
   b. Support random play of either all or a subset of all tracks in a MNG file
   c. Provide controls for volume and the commonly-used "threat" and "mood" variables

3. The advanced MNG player/editor shall
   a. Provide all features of the simple editor
   b. Provide a user interface allowing graphical editing of the MNG file, such that the user need not learn the scripting language
   c. Let users preview the music tracks

4. The MNG script parser component shall
   a. Accept a (decoded) MNG script or script fragment as input, and return a abstract syntax tree representation
   b. Throw suitable errors if an error in parsing occurs that
      i. Indicate where in the file the error was detected
      ii. Gives a best guess as to the nature of the error if possible

5. The MNG file filter component shall
   a. Provide an abstract interface for
      i. loading and saving MNG files
      ii. accessing and modifying the script
      iii. accessing, modifying, adding and removing samples
   b. Transparently decode and encode the script text

6. All programs and components shall
   a. Be written in fully CLS-compliant VB.NET or C#
   b. Conform to "best practices" in respect to design, commenting and user interfaces, as much as is possible given the time constraints
   c. Function intuitively, following standard metaphors for the user interface elements used
   d. Be capable of using all MNG files included in the game titles *Creatures 2, Creatures 3* and *Docking Station*
   e. Be provided with documentation appropriate to the intended users

Requirements Review

Having completed the requirements analysis phase, we had a reasonably complete specification to follow. At this point it was clear that there was a need for several components to fulfil the basic requirements:

- A low-level component capable of loading MNG files from disk and converting them into an internal representation for use by other components, and also of saving modified files to disk
- A scripting component able to parse a whole script file (and preferably portions of one) into an internal representation, and capable of generating a script from such a representation
- A component that worked with the scripting and file components and the operating system to play music
- A basic editor allowing the creation of new files, the addition and removal of samples and text editing of the script

Other desired end-user applications were a player with no editing functions, intended merely to load and play MNG files, and a graphical editor, aimed at allowing users to create modify MNG files without having to know the underlying scripting language.

Test Plan

The *Creatures* games came with several music files. Our plan was to test loading and saving these files and confirming that they were identical and that the parser verifier confirmed their validity, editing one by adding and removing samples and changing the script, and saving the result, then attempting to load it again.

Each file contained a number of tracks – about 40 tracks in total, and so to test the player component we proposed playing each of these tracks. However, each layer in each track could contain script instructions that were only set to trigger when variables such as "Mood" were at certain levels. It was therefore necessary to analyze each script to determine these trigger points and test these tracks multiple times with these variables set at appropriate levels to cover all paths. As the player component was designed with easily-accessible controls for these variables, this practice did not significantly increase the duration of testing, while uncovering bugs which might not otherwise have been found.

Design

It should be emphasised that the design of the programs involved was an iterative process. While the specification of tasks was reasonably clear, the most appropriate design for an interface or algorithm was not always so clear-cut; therefore after the initial throw-away prototype had been completed, used and discarded, the design was advanced through evolutionary development.

User Interface

Understanding that the underlying components are not what the user sees as the program (*"The user interface **is** the program"* – Alan Kay), our first thought on design was for the user interface. Having previously identified the required features, user interface design was a matter of deciding how these requirements could best be provided to the user with the tools available. Care was taken to follow the best practices of user interface while considering the

design, particularly with respect to discoverability, where the objective was not to require the use of documentation at any point.

MNGPad

We decided that a very simple interface would be most appropriate for the basic MNG editing applet, as the intended use was quick editing of MNG files by relatively advanced users, similar to the *Windows Notepad* accessory. Bearing this in mind, we named the tool MNGPad.

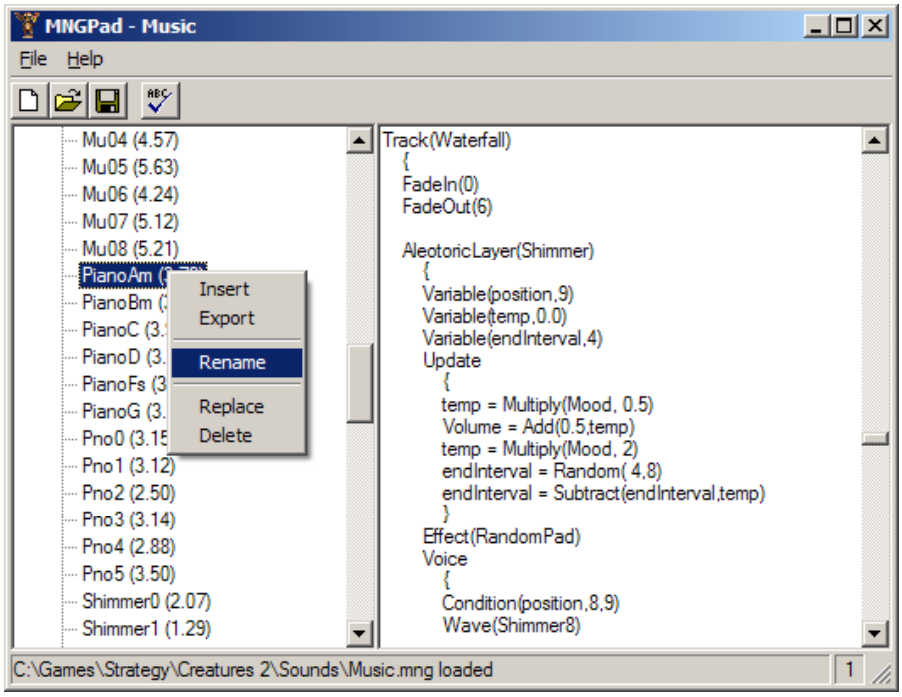


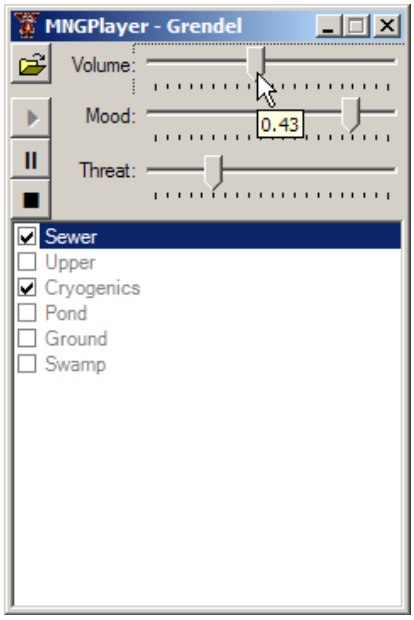
Figure 0-1: The MNGPad user interface

The MNGPad user interface consisted of two panes. The left pane contained a list of the samples present in the MNG file, the right pane held the script text. Menus at the top allowed the user to save, load or create new MNG files, and a single toolbar at the top offered access to commonly-used menu functions, as well as a button for basic validation of the script. This design reflects the limited features and intended use of the applet.

Formatting of the user's script was preserved by avoiding any conversions from the internal abstract syntax tree to text format, as we felt this was an important part of making the editor comfortable to use. Tool-tips were used to enhance discoverability for features such as script validation and the drag-and-drop functionality.

MNGPlayer

The MNGPlayer accessory also featured a relatively plain interface. Files could be opened via dialog box or dragged onto the player. Once loaded, the tracks were presented to the user as a list. Multiple tracks could be selected for simultaneous playback if desired. Playback could be started, paused, resumed and stopped by pressing the appropriate buttons.



The three main variables present in the scripts – Volume, Mood and Threat – were Creatures sliders which the user could drag to set a displayed as value; the precise value being displayed desired user dragged the slider thumb. As it was while the unobtrusive background playing, the applet designed for minimised to the system tray whilst still could be music. playing the

Figure 0-2: The MNGPlayer user interface

MNGEdit

MNGEdit was to be a more powerful and fully-featured editor, offering full drag and drop high-level editing of structures such as tracks, layers and effects, as well as the ability to play the music whilst it is being edited. The user interface design reflects this, expanding the sample list to a tabbed pane allowing the user to switch between the samples tree and lists of effects with properties, and replacing the script editing pane with a layered visual display of tracks.

MNG Parser Design

Early on we decided that the player component would parse the script file once and then work off an abstract syntax tree. This approach was considered more flexible and efficient than reparsing the tree from the script each time – parsing the tree once and working on an internal tree structure had only the initial cost, plus the result was easier to modify programmatically and output is a trivial matter of walking the tree; although it might be considered a waste to keep the tree in memory, in practice the size of the tree was not significant relative to the size of the samples.

Designing and building a full lexer/parser engine was beyond the scope of this project. Fortunately there was no need to do so, as the C# compiler tools (Crowe 2004) provided a comprehensive parser engine, taking as input a lexical and grammatical definition of the language, and outputting C# code to produce the abstract syntax tree.

Nodes

Syntactic elements in the script were represented internally by nodes. The base class returned by the parser was an instance of *Node*, and numerous subclasses of this class were created to describe the various expressions, effect, track and layer declarations and update assignments. Where appropriate these nodes contained lists of sub-nodes – for example, each *EffectNode* contained a list of *StageNodes*.

The Visitor Class

An abstract Visitor class (Gamma et. al. 1995) formed the basis for classes intended to walk the AST – for example, the various Reader classes mentioned below in MNGEngine. This class contained methods for each subclass of Node, intended to be called when the Visit method was called on that particular Node – for example, the Visitor.VisitTrack() method was invoked when TrackNode.Visit() was called. The default implementation of each method was empty, so that if a Visitor encountered a Node that it was not intended to deal with, it did nothing.

The MNG File Filter Component

The file filter component was designed to present an object-based front-end to a MNG file, offering both an in-memory representation as well as the ability to save and load files. Classes such as *Sample* that were intended to be used but not created by client applications had access to their constructors restricted with the Friend modifier. Similarly, clients were not permitted to directly add Samples to a *SampleHashtable*[4] but instead were forced to use a method which verified the integrity of the WAV files being added. Initially the component used a custom parser to detect the sample names (required because the names were only stored in the script) – later the design was changed to make use of MNGParser for this task.

Structure of the Music Engine

The music playing engine component (MNGEngine) was the most complex component in terms of design. Difficulties arose both from the inherent complexity of writing a script-based music playing component, as well as the lack of understanding surrounding the script at the beginning of the project. The details of the engine's design changed throughout the process of development, although the basic architecture remained relatively constant.

Readers and Players

It would have been possible to use one class for both reading from the tree and playing, but we felt that the clarity of the code would have suffered – it made sense to separate the two operations. For this reason the functionality was separated into *Reader* classes (which were subclasses of the *Visitor* class declared in the MNGParser component), and *Players*.

The Readers were responsible for reading the tree by calling the Node.Visit() method of the appropriate node, with themselves as the Visitor parameter. The nodes then called back the appropriate Reader, which either recorded a value (for those nodes which indicated attributes) or descended further into the tree. For example, *TrackReaders* created an array of *LayerPlayers*, adding one to the array each time processing reached the TrackReader.Visit[Aleotoric|Loop]LayerPlayer() method.

This data was stored in the Player – each Player created an instance of the Reader on initialization and then reads out the data. Once initialized, the Player.Play() method was called to begin playing the music. Player.Reset() stopped any current playback, and Player.Pause() temporarily halted it until called a second time.

[4] It should be noted that this protection is not totally secure – a client could bypass it by using .NET reflection to make any calls it desires (Brown 2004). We assume that if a developer chooses to do so, then they are prepared to deal with the consequences.

The Sound Manager

Sounds were handled by a special *SoundManager* class. The main objective of this was to make it easier for future porting efforts – in particular, porting to other platforms using a different sound output system. The SoundManager was initialized by the file-loading routine, using the Singleton pattern (Gamma et. al. 1995) to ensure that only one instance of the class would ever be present.

In an attempt to reduce the memory use, playback buffers handled by SoundManager were not created by the *VoicePlayer* class until the voice was played, and were removed when the sound was reset.

Threaded Playback Architecture

We believe the original *Creatures* music engine had run as part of a continuous loop within the game engine. Being a game, there was no problem in using up all available CPU time while running, a luxury not considered acceptable for this project. It was therefore decided to base the playback of sounds around a threaded architecture in an attempt to preserve interactivity while using as little CPU time as possible.

The .NET runtime provides a thread pool, to which client applications can delegate work items to be performed at a set time. However, this was not considered suitable for the execution of music scripts, because no facility was provided to suspend or abort threads, and the number of concurrent threads was bounded by the thread pool – there was therefore potential for exhaustion of this thread pool, and subsequent delay of processing. Instead, each layer was given a thread (later, a set of threads, if effects were in operation) – thus if a voice were followed by a pause it would not pause the other layers playing simultaneously.

A decision had to be made as to whether to have a thread sleep within the playback subroutine or to set a timer to complete the subroutine. We decided to use the threading sleep function because:

- It simplified the programming model
- It removed the need for yet another thread for the timer, which would be taken from the timer pool (with the previously-mentioned disadvantages)

Interpreter Design

Many of the music scripts relied on update processing. These updates could affect the volume, panning and interval between sounds, as well as passing on the results of calculations as variables to future updates. Certain variables (in particular, Mood and Threat) were designed to be altered by an external program and read by the script, the results varying the sounds played and/or the manner of their playback – for example, increased Threat might cause a decrease in the interval between notes, or cause a set of new samples to begin playing.

A question arose during design as to whether the evaluation mechanism for Add/Subtract/Multiply/RandomNodes should be within the interpretation and playback engine or within the tree component itself, as a GetValue() method that was part of the Node

objects. We decided that it was more appropriate for the engine to handle this task, as the tree was intended to be a passive data structure – "smart" nodes would defeat the point of this. Moreover, it would mean that future add operations might be restricted to those provided by the nodes. In the final design, only ConstantNode objects were defined to have an inherent value.

The expression evaluation process was operated as a stack machine. The base data type of the machine was the Single floating-point number. These values were pushed onto and popped off of an stack internal to the evaluator as appropriate to the nodes encountered during the evaluation. For example, if an Add instruction was encountered, the two sub-branches of the node would be visited – resulting in two values being pushed onto the stack – then the add operation was performed by popping these two values, using them as operands, and pushing the result onto the stack. Constants and variables formed the leaves of this tree, being elements that just pushed values onto the stack. In this way expressions of arbitrary complexity could be evaluated.

Implementation

Parser Construction

Our first main development task was to define the tree structure using the Visitor pattern. This resulted in one base Visitor class, and a set of classes deriving from a base Node representing the various semantic elements that could be present within a script – for example, an EffectDecNode represented an effects declaration, which contained a list of StageNodes, each of which had sub-Nodes relating to the settings for that stage.

We then proceeded to craft the input scripts to the lexer and parser generator, testing it with sample scripts that had been obtained using the prototype. At first, work on the parser was slowed by the need to perform external complication of the parser into C# by running a batch file. We found it possible to streamline the build cycle by incorporating the parser generation within the project through the use of pre-build steps in the Visual Studio environment. However, the parser generation still took over 20 seconds, an impediment to quick test cycles.

In an attempt to reduce the time taken, use was made of a parser generation flag to avoid generation of an LALR(1) parser and instead produce an SLR(1) parser, cutting generation time to less than two seconds. This increased productivity while retaining the ability to create a full LALR(1) parser if required.

Handling the Lack of Parametric Polymorphism

One disadvantage of .NET is that it currently does not support *generics*, also known as *parametric polymorphism* or *parameterization by types*. The functionality is provided by templates in C++, although the concepts and implementation are slightly different. Kennedy and Syme (2001) have extended this idea to the .NET platform in a runtime-efficient manner, unlike the Java implementation (Bracha et. al. 2001) which converts to the generic Object at runtime. Unfortunately this efficiency comes at the cost of incompatibility with previous versions of the .NET Common Language Runtime, and so will not be available until the CLR 2.0 release in late 2004.

As a result of this lack of generics, the generic list and collection classes provided by the .NET Framework and used throughout the project (such as *ArrayList* and *Hashtable*) operated on elements of type Object. This had both a performance and a type-safety impact. The performance impact was ameliorated by using the VB.NET *DirectCast* operation to avoid type coercions performed by the generic *CType* conversion function. The type-safety issue was attacked in several ways:

- Wrapping the class in such a way that only the appropriate types were accepted – for an example, see the *SampleHashTable* class in MNGFilter
- Checking the type of each element extracted from the class to ensure it was of the correct type – used by *SampleScanner*
- Checking that no code accessing the list was adding an unexpected type

General Issues

Thread Handling

The first attempt at threading the layers started a new thread when playing the layer and stopped it with Thread.Abort(). This worked but caused a noticeable delay on stopping the music, as aborting caused an exception to be thrown, and the main UI thread was required to wait for the abort to complete. This was avoided and both overall and perceived performance improved by altering the playback thread to wait on an event object when pausing between samples and exit if the event was signalled. The pause function was implemented in a similar manner, causing the threads to wait on an event object until un-paused.

It was also found necessary to boost the priority of the playback threads slightly to ensure that they received attention promptly upon being removed from the Sleep state when other tasks with high priorities were present. Since very little was done in each time-slice (typically just scheduling the playback of a single sound) before the processor was yielded by the thread, this did not generally interrupt the normal functioning of the system (but see section 7.4).

Secure File Handling

Library file operations such as Sample.Export() and MNGFile.Load() were initially designed to use filenames. However, we realised that this would preclude their use in situations where security is paramount. In this case, the use of files is usually restricted to opening file Streams via system file dialog boxes. For this reason these file-handling methods were converted to use the Stream class.

Global and Local Variables

Some variables are local to the layer currently executing. However, there was a need for globally modifiable variables such as “Mood”, “Threat” and a global volume control. To solve this, we implemented a *VariableHashtable* subclass of *Hashtable* that combined two sources – its base, and a second hashtable shared among all instances of this new class. This second hashtable was externally modifiable via the MNGEngine.GlobalVariables property. The engine could only make modifications to the first hashtable, but took variables from both

hashtables. In this way the global variables could be modified at runtime while the layers were playing, and changes were picked up immediately.

Packaging, Documentation and Distribution

Setup

Setup was implemented using the predefined setup project in Visual Studio. The tools were distributed in MSI format, the installer creating shortcuts to MNGPad, MNGPlayer and the MNGTools documentation on the user's Start menu.

Documentation

Many such projects fail to aid their intended user base, not because of a lack of coding skill on the part of the programmer, but because the users (who may themselves be skilled developers) cannot understand how to use the material they have been given. We realised at an early stage that comprehensive documentation for both users of current MNG applications and future developers would be vital.

User documentation was needed for the two user applications, MNGPad and MNGPlayer. MNGPlayer was relatively easy, but MNGPad required a reasonably detailed, yet approachable guide to creating music scripts.

Developer documentation covered the MNGTools suite, in particular MNGEngine, MNGFilter and MNGTree. Construction of the documentation was assisted by the use of an open-source project known as NDoc (NDoc 2004).

NDoc was designed to process the XML documentation produced by Visual Studio for C# into documentation, thus allowing developers to write documentation into their code modules, in a similar manner to Javadoc (Sun 2004b) - in fact, it can output to that format, among others. We wished to use NDoc to output HTML/Compiled HTML (.chm) documentation, providing developers a familiar help experience to that in Visual Studio. Unfortunately Visual Studio does not yet natively support XML documentation generation for VB.NET. However, an add-on called VBCommenter (GotDotNet 2004) was found to provide equivalent functionality and so was used to create the required XML files.

Hosting and Distribution

Hosting was provided by SourceForge (2003). The source code to the components and applications was distributed under the GPL, allowing fellow developers to create free tools based on our work.

Testing

Methodology

As previously covered in our test plan, testing was an easy matter in some respects – the *Creatures* games provided a reference implementation, with a variety of samples demonstrating the capabilities and limitations of the music system. In theory, all that was necessary was to ensure that the output was as expected given the input, and in general this was the case for tests involving the file and parser components and the editor.

In contrast, examining the results of auditory tests proved difficult as their output was in the form of sound intended to be sent straight to the speaker system. Moreover, the music was designed to have a random component, in addition to the input of in-game variables whose values could not be precisely known, and therefore it might be that one run produced significantly different output to another.

For these reasons, sound capture software was in general not used for testing – instead gross errors were diagnosed by ear, and where this was not possible specific test cases were constructed to compare the performance in game with that of the MNGEngine component.

For non-perceptive testing of the script interpreter, code was inserted into the components to write interpreter operations such as assignments and whether or not conditions for voice playback were being fulfilled when evaluated to a log. This allowed us to decide whether or not the code was proceeding down the correct path of execution, and in general aided the development of the engine. This advantage became even more apparent when threading was introduced, a development which caused single-stepping through execution to become impractical – debug statements could be tagged with the layer name and effect stage number, easing the task of identifying the sequence of operations.

Musical Output

No significant defects were found in the complex area of updates. However, while testing effects, it became clear that an error had been made, not merely in the implementation, but in interpreting the meaning of the script. The MNGEngine audio output simply did not match that of the game.

Initially we had thought the stages to be indicative of shifting the Pan and Volume settings for the affected layer's output from each voice, shown below as **(b)**. The first implementation of this was clearly jerky in comparison to the game, so we proceeded to try to smooth the output by interpolating between stages. However, this also produced incorrect output, and we soon realised that the engine was in fact playing copies of the same layer output, one copy for each stage.

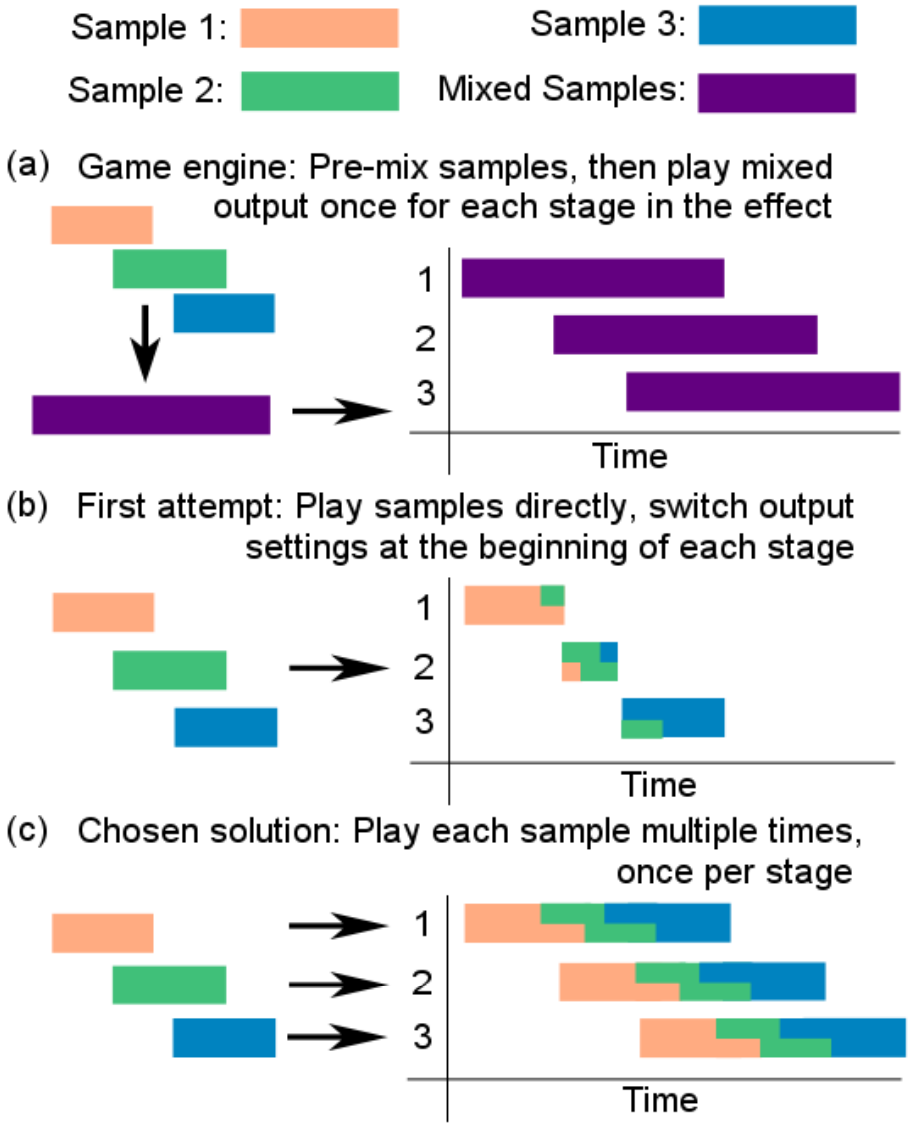


Figure 0-1: Comparison of effect handling methods

At this point, a difficult decision had to be made. One choice was to attempt the same as the original engine – to create a buffer filled with the output and then play it at the intervals specified by the stages. However, this would have required altering the model from playing samples directly to playing them to the output buffer and then playing the output buffer multiple times. As DirectSound does not offer access to this buffer we would have had to mix our sound, requiring the addition of mixing algorithms to the system, which we wished to avoid.

The other choice was to add extra delayed threads to play the buffers again. We decided to try this first, as we already had a threaded system to play layers. This approach ran into two problems, and although the music produced did have significant random quantities, was perhaps of academic interest, and could be the subject of further research, it was not the objective of this project.

Firstly, the threads used separate variable stores and evaluation routines to ensure that one thread did not interfere with the other; an example being a counter, which would be incremented once by each delayed thread. Unfortunately, the separate evaluation included separate random number generation, and combined with scripts defining intervals based on these numbers, output from the threads fell out of synchronization almost immediately. This issue was corrected by passing an identical random seed to each of the threads to be used in construction of a random number generator object. This caused evaluations to use the same sequence of random numbers and thus achieve the same results if changes to global variables over that time period were discounted; typically this was the case, as effects tend to last no longer than three or four seconds.

Another more subtle effect caused even those stages whose layers were not using random numbers to slowly drift out of synchronization. The underlying operating system is not a real-time guaranteed operating system, and so sleep operations may take longer than specified, particularly. This affected both the internal layer thread delays, causing notes to be played later than intended, as well as the delays between starting each layer-handler thread of an effect, causing the entire effect to be delayed more than intended.

Combined, the above problems caused unintentional delays of the order of 40 to 100ms; enough to significantly affect the tone of the music, given that some intended delays were no more than 200ms. These delays were corrected by introducing code to measure the actual time elapsed and subtracting this from the pause, rather than assuming that no time had passed except while sleeping. This also allowed the player component to recover if a high-priority task momentarily pre-empted it.

One final error was that the effect stages were computed only once, when starting the thread controlling the stage, whereas in the game they appeared to be computed each time the stage was run. This had no effect on the initial playback of the layer, but when repeated it became noticeable on those tracks that used the feature. The proposed fix involved moving such code to the thread handling the stage and running it each time.

While not an error, the musical output of the player was significantly louder than that in the *Creatures* games. We believe that this is either a deliberate reduction to reduce the impact of

the music, or a side-effect of gain-reduction performed by the engine in the mixing phase to avoid clipping. However, metaroom developers commented that this was in fact a help to them, as they could more clearly hear how their music was working. Users were able to reduce the volume with a slider.

Usability Testing

Initially, MNGPad was designed to create a new MNG file on opening. However, user testing quickly indicated that the first thing users did on loading the program was to open a music file for editing; therefore it was decided to automatically display the open file dialogue when starting the program, and allow the user to cancel this dialogue if they wished to create a new file. A similar feature was added to MNGPlayer.

In early iterations of the prototype tools it was found that the file load operation could take significant time on computers with slow hard disks, during which there was no feedback to the user. This was considered unacceptable and so a `LoadMNGProgress` event was added to the filter component and fired before loading each sample, which the client could then use to inform the user (for example, "Loading sample 3 of 24")

One external metaroom developer used the same name for different layers (by copying and pasting the text). This caused problems because the game engine was not designed to cope with this, although as the MNGEngine did not rely on the name of the layer, it was not affected. The problem could be fixed by adding semantic checks to the script, ensuring that layer names were distinct.

Users were slightly confused by the MNGPad validation errors, which gave line numbers as zero-based rather than one-based. Similarly, they felt that the caret should be positioned before the error, not after it. The required changes were simple but effective in increasing usability of the software.

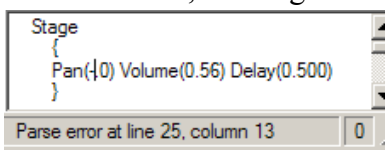


Figure 0-2: Users preferred cursors placed before errors

Other Testing Results

We initially believed that volumes would require only one node to represent them in the tree – after all, they were by themselves syntactically similar. However, when creating a test Visitor to write out the music scripts, it became clear that there was a difference between the two – the Volume attribute in a layer was typically printed on its own line, while the Volume in an effect's stage was usually printed without a new line. To store this context purely to identify one Volume node or another would have been against the principle of the code – such context could more easily be detected at the parser phase. We therefore decided that the Volume token would have two AST representations – the first for when it was detected in an effect declaration, the second for when it was an attribute of a layer. This representation allowed specific handling of both cases.

It was found that dragging MNG files onto the title bar of the applications did not always work. This was because the file was a shortcut (a link to another file), and while the file open dialogs dereferenced shortcuts, the drag-drop operation did not. The use of third-party code

was considered to handle these shortcuts, but could not be included as a result of licensing issues.

Testing the save function with read-only files revealed an issue – by their nature, read-only files cannot be written to, yet they are selectable in the file dialogs. MNGPad was modified to query the user and then attempt to unprotect the file if they chose to continue the save operation.

The multithreaded nature of the code required care in ensuring that thread-unsafe procedures were protected by synchronization mechanisms. In one case (the allocation of *SecondaryBuffer* objects in the *SoundManager* class) this was not done. This caused occasional index out of bounds errors, since one thread was creating a secondary buffer while another thread was accessing the sample's memory stream to do the same thing. The chosen solution was to wrap the location in a lock operation, such that the second thread had to wait for the first to complete its operations.

During testing with the "release" version of MNGPlayer we found that the *Creatures 2* Volcano track would cause 100% CPU usage when it was played with a certain Mood level. In this situation no sounds were played in one layer, and no interval was set, so the thread cycled without yielding. This was compounded by the fact that play threads were placed on a high priority to improve timing performance, thus denying the CPU to any other applications, as well as to the thread(s) of the controlling application (making it hard to close the program). The "debug" version did not exhibit this problem because it contained debugging output calls that yielded the CPU. This problem was only caught due to our practice of analysing the script to detect trigger values for variables and testing each case – it was not a typical situation. Our chosen solution was to yield for a minimum amount of time each cycle, ensuring a window of opportunity for other threads.

Conclusion

In this final chapter we reflect on our achievements, review critical design decisions made during the course of the project, give an brief overview of current user feedback, and offer suggestions for future work to build on our efforts.

Concrete Achievements

The stated goals of this project were to identify and document the MNG music format and create tools with which others could modify and enjoy music in this format. We have achieved these goals, creating the following concrete objects:

The MNG File and Scripting Format Specifications

A complete specification of both the on-disk layout of MNG files and the syntax and semantics of the scripting language is present in this text and its appendices, as is the algorithm required to decrypt the script. Previous to this project the file structure and general format of MNG files was completely unknown; we have provided the means for others to create programs using the format, should they deem the following tools provided by us to be inappropriate for their needs.

MNGPad – A Basic MNG File Editor

MNGPad fulfils the needs of metaroom developers by providing them the ability to examine and modify existing music scripts, add, extract and replace samples or create entirely new music files for their projects. It also provides a validation tool which may be expanded in the future to detect more subtle errors.

MNGPlayer – A Background Music Player

MNGPlayer allows users of the *Creatures* games to play the music without having to play the game and providing a means for metaroom developers to test their music files while the fully-featured MNGEdit is completed. It also serves as a demonstration of the use of the MNGEngine library for developers.

MNGTools – A Library for MNG File Management and Playback

The MNGTools package is composed of four separate components:

MNGFilter

Responsible for loading and saving MNG files, MNGFilter is used in both MNGPad and MNGPlayer. It handles the sample collection, which includes importing and exporting samples, as well as encoding and decoding the script.

MNGParser

The MNG script parser was initially intended solely for the use of MNGPlayer, handling the conversion of the music script file from plain text to the abstract syntax tree representation. However, user feedback demanded a way to validate scripts, and in any case it proved necessary to parse the script in order to identify the names of samples – MNGParser proved to be faster for this than the ad-hoc parser constructed for the prototype.

MNGEngine

As the player component of the suite, MNGEngine is responsible for taking a file and AST representation of the music script and turning this into audible sound. It is assisted in part by the DirectSound library, which handles music mixing and playback – MNGEngine schedules playback of the samples, which requires it to run an interpreter for the script updates as well as keep track of layer variables.
It is capable of playing multiple layers and multiple tracks in real time.

MNGTree

The MNG script tree library is used by the MNGParser and MNGEngine components. This component is not particularly complex, being mostly a library of node classes to be read and modified by other components, but it was essential to get it right early on – and in particular,

the Visitor class it contains, which is subclassed to make the various Reader and testing classes.

The tools are compatible with programming languages capable of using COM components, such as Visual Basic, C++ and Delphi, as well as with all .NET languages. As with the applications, they are released under the GNU GPL and available online. (MNGEdit 2004)

The MNGTools documentation should also be considered an achievement, as we believe it adds significant value to the project. Lack of documentation is a deficiency in many third-party tools – we consider it to be essential, particularly for developers who may not wish to read the code to understand what it does.

Review of Design Decisions

Several design decisions were made during this project. It was possible to make some early on; others only became apparent as details of the implementation were being considered.

Use of the Parser Generator

The decision was made early on to use an external parser generator rather than to attempt to parse the MNG script through ad-hoc methods. We believe that this was by far the better choice. Although requiring some research to setup and use the tools, the parsing system proved well-matched to the language, and the time spent was more than offset by the benefits of a relatively comprehensive, mature and well-tested lexer and LALR(1) parser generator.

One downside to the use of this parser was that both the generated parser and the parser runtime were quite large in comparison to the executables and other components. This was partially offset by the provision of a "runtime" parser library of reduced size – in addition, the parser proved highly compressible in distribution. In general, overall size was not a major concern considering the disk and memory requirements of the samples themselves, but the size of the installer was a concern for a program intended for online distribution.

Use of the Visitor Pattern

The second decision, to use the Visitor pattern on a tree representing the script structure, should also be considered a success. Use of the pattern turned the creation of new operations on the tree into a simple matter of subclassing the Visitor and overriding the necessary methods, and enabled the rapid creation of multiple tree readers – the sample file name scanner took ten minutes to implement and worked first time. It also allows for reuse of the MNG tree structure by other programmers. We therefore agree with the assertion that this is an appropriate design to use when constructing interpreters.

Choice of Language and the .NET Runtime

Having chosen .NET, there was in theory no restriction on the language used – all languages compile to compatible bytecode. However, for the purposes of ease of development the choices were restricted to those supported by the development environment. In effect, the choice of language made little difference – C# was required for the parser, while Visual

Basic.NET was chosen based on ease of development, and few problems were encountered in using either of them.

The .NET Framework is provided with the installation of the runtime and so is available to all .NET programs. Access to such an extensive class library meant that no time needed to be spent creating or debugging hash tables, file I/O algorithms or the like – only code germane to the project was required. This led to increased productivity and so enabled the implementation of a rich feature set.

The most pressing concern was performance – garbage collected languages have a reputation of problems in this area. However, this concern proved unfounded, as garbage collection rarely took more than 0.5% of process CPU time. We believe current garbage collection algorithms are well-equipped to handle almost all types of application, as long their performance characteristics are considered when in use. Moreover, much of the work in mixing and output of samples could be handed off to the native libraries which in turn delegate to the sound hardware, further boosting performance – in our tests the player tool less than 5% of CPU time on most tracks, and was even able to play all 19 music tracks in the *Creatures 2* music library simultaneously, while leaving half the CPU time idle.

A more pressing issue was the requirement for the .NET runtime and libraries on the user's machine. There was considerable resistance from some users to download a 20Mb system update for no clear benefit. In retrospect this could have been anticipated – many home users are still on modems, for which this is almost an hour's download. Overall, though, we believe the language and .NET runtime were a suitable choice despite this disadvantage. It would not have been possible to achieve as much in the allocated time without it.

Use of DirectSound

DirectSound offered a mature, low-latency output mechanism and few difficulties were encountered when using it. CPU usage was in general very low, although some of this can be credited to the advances in sound technology – current hardware is well-suited to the task of playing multiple streams of sound.

The use of DirectSound has perhaps limited the portability of the solution, but the code using it has been isolated into the *SoundManager* class to allow the future development of a replacement, and so we believe this disadvantage is minimal. Indeed, its model of sound output based on buffers should be portable, as this paradigm is common among sound libraries.

Choice of a Multithreaded Effect Architecture

During testing it became clear that the support for effects was incorrectly implemented because of a misunderstanding of the meaning of the script. Fixing this to match the game would have required a copy of the post-mixed sound stream for repeated playback. Unfortunately DirectSound did not give access to the output of its mixing process for capture, and so this made it impossible to dynamically redirect the sound. Mixing sounds was against one of the objectives of this project, but the trivial algorithm is not particularly complicated to implement or CPU-intensive, and so it was considered as a serious option.

This choice was driven by time pressures. Given sufficient time, it is probable that we would have chosen to pre-mix the sounds in a similar way to the original game engine, to reduce the potential load on the CPU and hardware sound buffers. However, at the time the problem was discovered, making drastic changes to the structure of MNGEngine could have jeopardised its completion date. The layers were already being played by threads, and adding another to start new layer threads at set periods was a relatively simple change which did not require altering the way in which sounds were played.

In retrospect, this problem highlights the importance of a proper specification, and the cost of changes to that specification and design late in the project. In this case the specifications were sufficiently unclear to make this a hard problem to catch, but additional testing of the specification by constructing test cases in MNGPad before starting to implement MNGEngine might have caught the problem earlier.

Review of Implementation and Packaging

Music Playback Engine

In general we feel that MNGEngine is a good solution to the objective of being able to play MNG files. It does not use up excessive CPU time in processing the MNG files, although it relies on the ability of the underlying framework and operating system to create and dispatch threads quickly, which may not be possible in all circumstances. Its main flaw is that it can consume a very large amount of memory, although this is dependant on the tracks that the user chooses to play. While to an extent this is unavoidable considering the nature of the music, the situation could certainly be improved.

MNGTools Developer Documentation

The documentation process added significantly to the time and effort required to complete the tools, but we believe that strong documentation is essential if code is to be used by others who may not be familiar with the issues. NDoc streamlined the build process, and enabled output of professional quality in terms of look-and-feel with little effort. Documentation activities also forced us to consider how other developers might wish to use our work, sometimes leading to architectural changes to make the component interfaces more developer-friendly.

User Feedback

Initial user feedback has been very positive, both in anticipated and unanticipated ways. As expected, metaroom developers were eager to customize the user experience by adding sounds to their projects, and we have already received two custom music files intended for new metarooms (one user's quote: "*This music really adds something more to the room – it's got the right atmosphere now").* Others simply enjoyed listening to the music outside of the game, without the game itself slowing down their computers. One developer has begun to use the specification provided in this document to create their own programs, while another is working on the published source code. Unanticipated uses included exporting the samples for use in *Creatures*-themed web sites, and importing favourite *Creatures 2* music into *Docking Station*.

Future Work

MNGEdit

The stated goal of MNGEdit was the ability to edit MNG files without knowledge of the underlying scripting language. As such, it was envisaged as an an ideal interface for those users who are not interested in delving into complex scripts, and would therefore find MNGPad unwieldy or unusable. Unfortunately it was not possible to realise this program within the time scope of this project. However, we believe that with a solid music playback and file handling library, the future development of such a tool is not unfeasible.

Development of MNGPad

We assert that MNGPad is sufficient for the creation of any MNG file. However, there is much that could be done to improve its usability and functionality without compromising its objective of being a fast text-based editor. Future developments could include syntax highlighting, searching, and more specific error messages for syntax error handling (which would require work on MNGParser to support error productions), and possibly semantic checks for errors such as duplicate layer definition names or declaring a Voice using a Wave that does not exist.

Extended User Usability Testing

For the most part, testing on this project was restricted to confirmation of the correctness of functionality. We are well aware that developers are not able to fully test the usability of their own work, and although some remote testing was performed by sending the distribution to power users and eliciting their comments, we feel that more testing aimed primarily at usability targets would reap rewards. Getting non-technical users to attempt to create their own music could be an excellent way of testing the quality and appropriateness of the editing/playback applications and their supporting documentation.

Development of the MNGTools suite

Improvements to the user applications will necessarily drive innovation in the underlying tools library. As previously mentioned, the parser could be modified to handle error productions, allowing it to be more resilient in the face of user error. The playback engine might be altered to implement the premixed-buffer approach to effects, which could significantly reduce memory requirements. Alternatively, the loading of samples could be delayed, or the samples could be stored in so-called *weak references* that do not prevent the objects they refer to from being garbage-collected when necessary. Additional effects could be added to the language to directly leverage the abilities of the DirectSound library, although these would of course not be available to those creating music for the *Creatures* games. Proper support for externally-modified variables (a-la Mood and Threat) could also be added.

Portability

Portability was not a stated objective of this project, since only one of the games concerned ever ran on a non-Windows platform and the vast majority of target users run Windows as

their primary operating system. However, in consideration of the fact that many *Creatures* developers use Linux, and that the "Rotor" .NET runtime supports FreeBSD and Mac OS X, it would be desirable to have the MNGTools suite run under multiple platforms.

A portable sound library would be the main component required, and the Simple DirectMedia Layer (SDL 2004) might be one route towards this. Compatibility with the Mono .NET runtime would also need to be tested – at the time of writing, certain necessary portions of the System.Windows.Forms library have not been ported to it, and from our own testing with MNGPad we would agree that it "is not ready for use in a production system at this time" (Calvert 2004).

However, the parser generator is compatible with Mono, and we were successful in both loading and saving a MNG file and extracting a music script from it using a test program compiled in Linux against the MNGFilter/MNGTree/MNGParser libraries, which had been compiled in Windows. We are confident that more comprehensive support for Forms-based programs will develop within the next two years.

Bibliography

(Ananian et. al. 1999) Ananian, C. S., Flannery, F. and Hudson, S. CUP Parser Generator for Java – project web site: http://www.cs.princeton.edu/~appel/modern/java/CUP/

(Brooks 1995) Brooks, F. P., "The Mythical Man Month, Anniversary Edition (2nd ed.)", Addison-Wesley, 1995

(Bracha et. al. 2001) Bracha, G., Cohen, N., Kemper, C., Marx, S., Odersky, M., Panitz, S., Stoutamire, D., Thorup, K., Wadler, P., "JSR-00014: Adding Generics to the Java Programming Language", April 2001 [online source as of May 2004: http://jcp.org/aboutJava/communityprocess/review/jsr014/index.html]

(Brown 2004) Brown, K., "Beware of Fully-Trusted Code", Security Briefs, *MSDN Magazine*, April 2004 [online source as of May 2004: http://msdn.microsoft.com/msdnmag/issues/04/04/SecurityBriefs/default.aspx]

(Burns 1997) Burns, K., "History of electronic and computer music including automatic instruments and composition machines," Florida International University [online source as of May 2004: http://eamusic.dartmouth.edu/~wowem/electronmedia/music/eamhistory.html]

(Calvert 2004) Calvert, C., "WinForms: How to Use Mono to Get Microsoft .NET GUI Based Applications Running on Linux", Borland Developer Network [online source as of May 2004: http://community.borland.com/article/0,1410,32073,00.html]

(Chowning 1973) Chowning, J., "The synthesis of complex audio spectra by means of frequency modulation", *Journal of the Audio Engineering Society*, vol. 21, pp. 526-534, 1973

(Crowe 2004) Crowe, M. K., "Compiler Tools in C#", May 2004 [online source as of May 2004: http://cis.paisley.ac.uk/crow-ci0/index.htm#Research]

(Csound 1999) Boulanger, R. ed., "The Csound Book – Perspectives in Software Synthesis, Sound Design, Signal Processing and Programming", MIT Press, October 1999 [related website: http://www.csounds.com/]

(Clark 2001) Clark, A., "Adaptive Music", *Gamasutra Audio Resource Guide*, May 2001 [online source as of May 2004: http://www.gamasutra.com/resource_guide/20010515/clark_01.htm]

(Culver 2001) Culver, A., "John Cage Computer Programs", 2001 [online source as of May 2004: http://www.anarchicharmony.org/People/Culver/CagePrograms.html]

(Cyberlife 2000) CyberLife Technology, "The Music Behind Creatures", 2000 [online source as of May 2004: http://web.archive.org/web/20020614201723/http://www.creatures.co.uk/library/science/lib_science_musicbehind1.htm]

(Cyberlife 2004) CyberLife Technology, "Creatures 2 CAOS Language Guide", pp17-18, February 2004 [online source as of May 2004: http://www.gamewaredevelopment.co.uk/downloads/COBcompiler/C2CAOS.doc]

(Delahunt 2003) Delahunt, M., "ArtLex on Aleatory", ArtLex Art Dictionary, 2003 [online source as of May 2004: http://www.artlex.com/ArtLex/a/aleatory.html]

(DotGnu 2003) DotGNU, DotGNU Portable.NET project website – http://dotgnu.org/pnet.html

(Gameware 2004) The Creatures Development Network website – http://www.gamewaredevelopment.co.uk/cdn/

(Gamma et. al. 1995) Gamma, E., Helm, R., Johnson, R. and Vlissides, J., "Design Patterns: Elements of Reusable Object-Oriented Software," Addison-Wesley, 1995

(GNU 2003a) The *flex* Home Page, Free Software Foundation – http://www.gnu.org/software/flex/

(GNU 2003b) The *bison* Home Page, Free Software Foundation – http://www.gnu.org/software/bison/

(GotDotNet 2004) The *VBCommenter Powertoy* Development Home Page - http://www.gotdotnet.com/Community/Workspaces/workspace.aspx?id=112b5449-f702-46e2-87fa-86bdf39a17dd

(Grand 1997) Grand, S., Cliff, D., Malhotra A., "Creatures: Artificial Life Autonomous Software Agents for Home Entertainment", *Proceedings of the First International Conference on Autonomous Agents*, ACM Press, New York, 1997, pp. 22-29 [online source as of May 2004: http://www.cyberlife-research.com/articles/grandcliffmalhotra/agents97.html]

(Hays 1998) Hays, T., "DirectMusic For The Masses", *Gamasutra Audio Resource Guide*, November 1998 [online source as of May 2004: http://www.gamasutra.com/features/19981106/hays_01.htm]

(IDA 2004) IDA Pro Disassembler website – http://www.datarescue.com/idabase/

(Johnson 1975) Johnson, S. C., "Yacc – yet another compiler compiler", *Computing Science Technical Report 32*, AT&T Bell Laboratories, Murray Hill, N. J., 1975

(Kennedy and Syme 2001) Kennedy, A. and Syme, D., "Design and Implementation of Generics for the .NET Common Language Runtime", Microsoft Research, May 2001 [online source as of May 2004: http://research.microsoft.com/projects/clrgen/generics.pdf]

(Klein 2003) Klein, G., "JFlex – The Fast Scanner Generator for Java", [online source: http://www.jflex.de/]

(Land and McConnell 1991) Land, M. Z. and McConnell, P. N., "Method and apparatus for dynamically composing music and sound effects using a computer entertainment system", November 1991, U.S. Pat. 5,315,057

(Lesk 1975) Lesk, M. E., "Lex – a lexical analyzer generator", *Computing Science Technical Report 39*, AT&T Bell Laboratories, Murray Hill, N.J, 1975

(Mathews 1969) Mathews, M. V., "The Technology of Computer Music", MIT Press, 1969

(Maurer 1999) Maurer, J. A., "A Brief History of Algorithmic Composition", 1999 [online source as of May 2004: http://ccrma-www.stanford.edu/~blackrse/algorithm.html]

(McKinney 1997) McKinney, B., "Hardcore Visual Basic (2nd ed.)", pp109-113, Microsoft Press, 1997 [online source as of May 2004: http://www.mvps.org/vb/hardcore/]

(IBM and Microsoft 1991) IBM Corporation and Microsoft Corporation, "Multimedia Programming Interface and Data Specifications 1.0", August 1991 [online source as of May 2004: http://www.tsp.ece.mcgill.ca/MMSP/Documents/AudioFormats/WAVE/Docs/riffmci.pdf]

(Microsoft 2002) Microsoft Corporation, "The Microsoft Shared Source CLI Implementation", March 2002 [online source as of May 2004: http://msdn.microsoft.com/library/en-us/Dndotnet/html/mssharsourcecli.asp]

(Microsoft 2003a) Microsoft Corporation, "Overview of the .NET Framework" in *.NET Framework Developer's Guide* [online source: http://msdn.microsoft.com/library/en-us/cpguide/html/cpovrIntroductionToNETFrameworkSDK.asp]

(Microsoft 2003b) Microsoft Corporation, "Microsoft DirectX Audio Overview" [online source: http://msdn.microsoft.com/library/en-us/dnaudio/html/daov.asp]

(MIDI 2003) MIDI Manufacturers Association website – http://www.midi.org/

(MNGEdit 2004) "MNGEdit – a music editor for Creatures", project website - https://sourceforge.net/projects/mngedit/

(Morton 1990) Morton, J., "Atari Advertises the STacey, Commodore Fights for MIDI Acceptance" in *BetaZine #8*, The PsychoTronic Authority, April 1990 [online source as of May 2004: http://www.atariarchives.org/cfn/12/03/0041.php, picture of advert: http://www.atari-explorer.com/images/ST-midi-Ad1-Small1.gif]

(Mono 2003) Ximian, Mono project website – http://www.go-mono.com/

(NDoc 2004) NDoc project website – http://ndoc.sourceforge.net/

(SDL 2004) Simple DirectMedia Layer website – http://www.libsdl.org/

(Smith 1999) Smith, S. W., "The Scientist and Engineer's Guide to Digital Signal Processing", pp261-350, California Technical Publishing, 1999 [online source as of May 2004: http://www.dspguide.com/pdfbook.htm]

(SourceForge 2003) SourceForge website – http://sourceforge.net/

(Sun 2004a) The Java Sound API website – http://java.sun.com/products/java-media/sound/

(Sun 2004b) The Javadoc Tool website – http://java.sun.com/j2se/javadoc/

(Thompson 2003) Thompson, T., "The PLUM list – Programming Languages Used for Music", website - http://www.nosuch.com/tjt/plum.html

(Whitmore 2003) Whitmore, G., "Designing With Music In Mind: A Guide to Adaptive Audio for Game Designers", *Gamasutra Audio Resource Guide*, May 2003 [online source as of November 2003: http://www.gamasutra.com/resource_guide/20030528/whitmore_01.shtml]

(Wikipedia 2004) "One-time pad", *Wikepedia*, Februrary 2004 [online source: http://en.wikipedia.org/wiki/One-time_pad]

(Yackley 1999) David Yackley, "Microsoft DirectMusic: Creating New Musical Possibilities", Microsoft Corporation, November 1999 [online source: http://msdn.microsoft.com/library/en-us/dnmusic/html/dm_nmp.asp]

(Zend 2003) Zend Accelerator product website – http://www.zend.com/store/products/zend-accelerator.php

MNG File and Scripting Formats

This appendix is intended as a reference to assist programmers in writing their own MNG-based programs in any language, not necessarily using the tools developed by us. It is divided into two parts:

1. A description of the layout of MNG files on disk, and
2. The syntax and semantics of the script included in the MNG file

Unless otherwise specified, all lengths are in bytes, positions are 0-based from the start of the file, and numerical values are little-endian. This information was obtained via reverse-engineering techniques and so is not authoritative.

MNG File Format

Table A-1: MNG File Disk Layout

Position	Length	Description	Notes
0	4	Number of samples	
4	4	Position of script	Zero-based byte position
8	4	Length of script	Length in bytes
12	4	Position of first sample	Zero-based byte position
16	4	Length of first sample	Length in bytes
20	4	Position of second sample	Zero-based byte position
…	…	…	
N * 8 + 8	4		N is the number of samples
N * 8 + 12	Variable	First sample	Followed by the rest of the samples

The samples themselves are in the WAV file format, except that the RIFF header is missing and the "fmt_" chunk is missing its identifier (that is, the first data present are the "length of format chunk", which typically starts at the 16th byte of a WAV file). Implementers should be aware that the file format does not appear to support any RIFF chunks other than the format and data chunks – when importing WAV files, additional chunks should be stripped out. The standard format of WAV files used in MNG files is 22050 Hz mono 16-bit audio – other formats are not guaranteed to work correctly.

The names of the samples as referenced by the script are not stored with the samples – they are implicitly defined by the position of the sample in the MNG file. The first sample to be named in the script with a Wave() definition will be the first sample in the MNG file; the second will be the second, and so on. Repeated samples are reused.

The script is ASCII text, but it is encrypted with an XOR function. This function works on a byte level, with a starting operand value of 0x5 and an increment of 0xC1. The following sample function will encode or decode an array of bytes into this format:

```
Private Function Scramble(ByVal data As Byte()) As Byte()
      Dim hb as Byte, count as Integer
      hb = 5
      For count = 0 to data.Length - 1
            data(count) = data(count) Xor hb
            If hb < &H3F Then
                  hb = CByte(hb + &HC1)
            Else
                  hb = Cbyte(hb + (&HC1 - &H100))
            End If
      Next count
      Return data
End Function
```

MNG Scripting Format

MNG script files are defined by the following lexical analysis script and context-free grammar (see Chapter 3 of the main text and the MNGPad user guide for semantics):

Lexical Grammar

Table B-1: Script Lexical Grammar

[\n\r]	;
Variable	%T_Variable
Effect	%T_Effect
Track	%T_Track
Stage	%T_Stage
Pan	%T_Pan
Volume	%T_Volume
Delay	%T_Delay
TempoDelay	%T_TempoDelay
Random	%T_Random
FadeIn	%T_FadeIn
FadeOut	%T_FadeOut
BeatLength	%T_BeatLength
AleotoricLayer	%T_AleotoricLayer
LoopLayer	%T_LoopLayer
Update	%T_Update
Add	%T_Add
Subtract	%T_Subtract
Multiply	%T_Multiply
Divide	%T_Divide
SineWave	%T_SineWave
CosineWave	%T_CosineWave
Voice	%T_Voice
Interval	%T_Interval
Condition	%T_Condition
BeatSynch	%T_BeatSynch
UpdateRate	%T_UpdateRate
Wave	%T_Wave
[A-Za-z]([A-Za-z0-9])*	%name
-?[0-9]+\.[0-9]+	%number
-?[0-9]+	%number
[(){},=]	%TOKEN
\/\/.*	%comment

Parser Grammar

```
goal:       statementlist ;

statementlist:      statement
| statementlist statement
;

statement:    effectdec
| trackdec
| variabledec
| comment
```

```
;

variabledec:        T_Variable '(' name ',' expression ')' ;

effectdec:    T_Effect '(' name ')' '{' stagelist '}' ;

trackdec:     T_Track '(' name ')' '{' track '}' ;

stagelist:    stage
| stagelist stage
;

stage:              T_Stage '{' stagesettinglist '}'
| comment
;

stagesettinglist: stagesetting
| stagesetting stagesettinglist
;

stagesetting:       pan
| effectvolume
| delay
| tempodelay
| comment
;

pan:          T_Pan '(' expression ')' ;

layervolume:        T_Volume '(' expression ')' ;

effectvolume:       T_Volume '(' expression ')' ;


delay:              T_Delay '(' expression ')' ;

tempodelay:   T_TempoDelay '(' expression ')' ;


random:             T_Random '(' expression ',' expression ')' ;


track:              tracksetting
| track tracksetting
;

tracksetting:       aleotoriclayerdec
| looplayerdec
| fadein
| fadeout
| beatlength
| volume
| comment
;

fadein:             T_FadeIn '(' expression ')' ;

fadeout:      T_FadeOut '(' expression ')' ;
```

```
beatlength:   T_BeatLength '(' expression ')' ;

aleotoriclayerdec:   T_AleotoricLayer '(' name ')' '{' aleotoriclayer '}' ;

looplayerdec:       T_LoopLayer '(' name ')' '{' layer '}' ;

aleotoriclayer:           aleotoriclayercommand
| alotoriclayer aleotoriclayercommand
;

looplayer:          looplayercommand
| looplayer looplayercommand
;

aleotoriclayercommand:    effect
| comment
| layervolume
| variabledec
| updateblock
| voiceblock
| beatsynch
| updaterate
| interval
;

looplayercommand:   comment
| layervolume
| variabledec
| updateblock
| beatsynch
| updaterate
| wave
| interval
;

effect:             T_Effect '(' name ')' ;

updateblock:        T_Update '{' assignmentlist '}' ;

assignmentlist:     assignment
| assignment assignmentlist
;

assignment:   variable '=' expression
| comment
;

variable:     name
| T_Interval
| T_Volume
| T_Pan
;

expression:   add
| subtract
| multiply
| divide
```

```
| sinewave
| cosinewave
| random
| variable
| number
;


add:          T_Add '(' expression ',' expression ')' ;

subtract:     T_Subtract '(' expression ',' expression ')' ;

multiply:     T_Multiply '(' expression ',' expression ')' ;

divide:       T_Divide '(' expression ',' expression ')' ;

sinewave:     T_SineWave '(' expression ',' expression ')' ;

cosinewave:   T_CosineWave '(' expression ',' expression ')' ;

voiceblock:   T_Voice '{' voicecommands '}' ;

voicecommands:      voicecommand
| voicecommands voicecommand
;

voicecommand:       wave
| interval
| effect
| condition
| updateblock
;

interval:     T_Interval '(' expression ')' ;

condition:    T_Condition '(' variable ',' number ',' number ')' ;

beatsynch:    T_BeatSynch '(' expression ')' ;

updaterate:   T_UpdateRate '(' expression ')' ;

wave:         T_Wave '(' name ') ;
```

MNGPad/MNGPlayer User Guide

[Note: This guide is aimed at end-users not academics, and is written accordingly]

Introduction

Welcome to MNGTools! First of all, if all you want to do is to listen to your *Creatures* music, then read no further – all you have to do is start up MNGPlayer, open the music file of your choice, select the track you want to listen to, and click play. These files are in the Sounds subdirectory for each game, with names ending with .MNG

If you want to learn more about how the music in *Creatures* works, or maybe make or edit music files for your *Creatures* games, read on . . .

How A MNG Music File Works

You may be familiar with music in the form of one long stream of sound, as in a WAV or MP3 file. However, music in the *Creatures* games works a little differently. It is made up small *samples* of music, usually just a few seconds long.

These samples are put together in various ways, as indicated by a script. This script is little like a piece of sheet music - it tells the game what samples to play as part of the music, and when and how it should play them. However, it is also a little like a programming language – it has loops, and variables, and operations such as addition and multiplication.

Don't worry if this all sounds terribly complicated – it's really not that hard to get to grips with, especially after you've seen a few examples!

Using Samples

As you have learnt, music in *Creatures* is made up of snippets of sound called *samples*. These samples are typically a few notes from a musical instrument, but they may be any sound you can devise, as long as you can get it into a WAV file.

The samples should be in 22 kHz (22100 Hz) mono 16-bit format. Your sound program should be able to save in this format; if it cannot, you can convert existing files by opening them in Sound Recorder and selecting this format in the Options dialog of File/Save As…

In MNGPad you may add, rename, export, replace or delete samples from music files from the menu that appears when right-clicking on them in the sample list. You can also drag and drop files from Windows Explorer into the sample list, or right-click and drag samples out into a folder. If importing files, MNGPad will use the name of the WAV file, but you are free to rename the sample later on.

One warning – the music file can only store the names of samples that you use, so you **must** use all samples by referring to them as a Wave (see below). If you do not, MNGPad may

warn you on exiting that the names of unused samples may be lost (or just not let you exit without removing them or altering the script).

Making Scripts

Tracks, Layers, Voices and Updates

When you specify music in *Creatures* or *Docking Station*, you specify a Track. These tracks are filled with Layers. Each layer can be thought of as a player in an orchestra, while the Track would be the orchestra itself. As you might expect, layers play simultaneously, but you only play one track at a time.

There are two types of layer, the LoopLayer and the AleotoricLayer. LoopLayers are useful if you only wish to play one sample, but you wish to play it all the time, maybe moving it from side to side on the speakers or fading it in and out. AleotoricLayers are more flexible, and can be used to play a sequence of Voices, each of which has a different sample.

Both layers have the concept of an Update block. Updates happen at a set rate (the UpdateRate) for LoopLayers, or each time you go through an AleotoricLayer. When updating, a series of assignments to variables – boxes inside the computer that contain a number – are made. These variables can be separate variables declared in the layer – like this: Variable(temp, 0.0) – or they can be preset things like Volume or Pan – the two most important ones. Volume ranges from 0.0 to 1.0, and Pan from -1.0 (full left) to 1.0 (full right).

The game also supplies two variables which you can read from, Mood and Threat. Mood is how well the creatures are, and Threat is how dangerous the situation is – if a grendel is on the prowl, you can bet that Threat will rise! You can see several examples of how variables are used in the music scripts included with *Creatures*, and test out changing Mood and Threat with MNGPlayer.

An Interval can be set for a layer. This is how long it will pause between voices (or between playing the same wave, for LoopLayers). You can also specify an interval in a voice, which will be used if that voice is played. Often intervals are timed so that a new sample starts just before or just after the previous one (although effects can be used for this as well). You can also set an update block for a voice, to be executed when that voice has played.

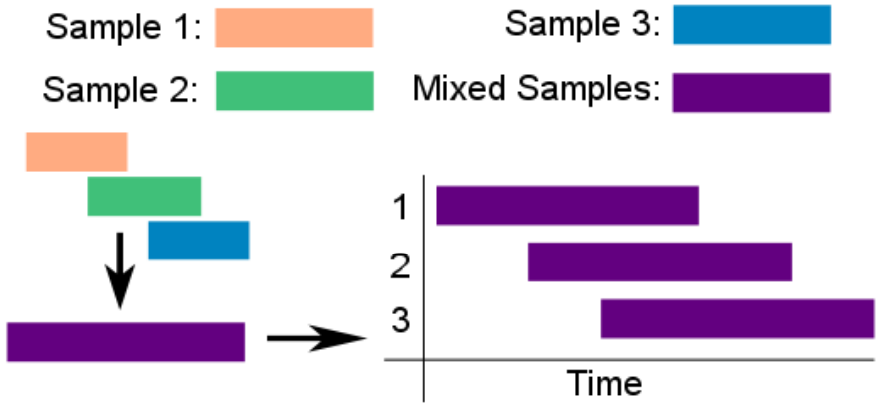
A voice may specify one or more Conditions which must be passed for it to play. For example it may only play if the Mood is between 0.5 and 1.0, or if a counter variable declared by the script is at a certain level. Conditions also affect whether or not update blocks and intervals contained within a voice take effect.

Using Effects

We've covered a lot, and if you've been following along in MNGPad you should be able to hear things taking shape, but perhaps it doesn't sound quite like the in-game music – a lot of the tracks have an "echoing" sound, or one that bounces from one speaker to another – somewhat like a LoopLayer. You may also be interested by all the text at the top of the

Creatures script files. These declarations are called Effects, and they control the output of each AleotoricLayer.

What do we mean by output? Well, you can think of it like the following diagram:



We start off with three samples. These samples are mixed together when we play each of them in a layer. However, the *Creatures* sound engine then takes that mix and plays it several times, once for each stage in the selected effect (or just once, if you haven't set an effect). In this case, there were three stages, so it played the entire mix three times.

The trick is that the engine doesn't play them all at once – each stage has a delay associated with it, the delay between that stage playing and the next one. They also have Pan and Volume settings, so you can make the sound slowly fade out or sweep from left to right, just like the *Creatures* music does. Try it out for yourself by applying an effect; all you have to do is add Effect (effectname) into the layer. Effects are usually defined at the top of the file, and are shared among all layers.

Hopefully Helpful Hints

Of course, this is all very theoretical - the best way to learn is to try stuff out! Using MNGPlayer you can quickly get an idea of how your music is going to sound – just save it in MNGPad and reload in MNGPlayer to listen to the changes. Note that you can drag and drop MNG files onto both MNGPad and MNGPlayer.

First of all, the music in *Creatures* is meant to be played all the time. This means that your music should be designed to run cyclically. At the same time, it may only play for a few seconds as the user moves on to another part of the world, and so it should probably not take too long to get going, unless you expect players to spend a long time in the same place. Bear this in mind while making your music. The game scripts are often good examples – listen to them and then read the script to see how they did it.

Talking of other scripts, it is often a good idea to start with an existing script so that you have the correct structure – just remember not to include anyone else's script (or samples) without their permission in music you distribute to others, since in almost all places it's against the law. Regarding music originally from the *Creatures* games, Gameware's view appears to be

that it **is** OK to use their work as long as you are **not** using it commercially – if you want to sell stuff including music, make your own! Like any copyright holder, they have the right to say no in specific cases if they wish to.

Finally, remember, computers are stupid – if you get a) in the wrong place, or miss out a {, it won't work! The tick button on the MNGPad toolbar will check that you've got them all in the right places, although it can't check everything – make sure to test your music first in MNGPlayer. And finally, the case of letters matters – SampleA is a different sample to samplea. Don't use both unless you want to get horribly confused.

Conclusion

That's it. Happy editing! If you have any suggestions, comments, or bug reports, send me an email at greenreaper at hotmail dot com. If you've done a really cool piece of music, I'd love to hear it, but send it to greenreaper at stardock dot com.

User Questionnaire

The following questionnaire was sent to users to quantify their experience of the prototype MNG editor, and to elicit suggestions from them:

Thank you for taking the time to try out this software. Please answer the following questions as best you can. Your answers will help us to modify the programs so that they are a better fit to your needs.

Why are you interested in this software? What do you imagine/intend using it for?

Having used it for a few minutes, what was your first reaction to the software?

What do you not like about how the current features work?

What features do you feel are missing?

If a future editor was made that did not require you to edit the script directly, what features would you consider to be most useful to you, and how would you imagine them working? Please give as much detail as you can.

Is there any other way in which the software itself, installation or documentation could be improved for you?

Do you have any other comments?

Scramble Function

This appendix documents the MNG scramble function as originally disassembled from the *Docking Station* game executable, engine.exe:

```
.text:00543400 sub_0_543400    proc near               .text:00543400
.text:00543400 arg_0           = dword ptr  8
.text:00543400 arg_4           = dword ptr  0Ch
.text:00543400
.text:00543400                 push    ebp
.text:00543401                 mov     ebp, esp
.text:00543403                 push    esi
.text:00543404                 mov     esi, [ebp+arg_4]
.text:00543407                 xor     eax, eax
.text:00543409                 mov     cl, 5
.text:0054340B                 test    esi, esi
.text:0054340D                 jle     short loc_0_543424
.text:0054340F                 mov     edx, [ebp+arg_0]
.text:00543412                 push    ebx
.text:00543413
.text:00543413 loc_0_543413:   ; CODE XREF: sub_0_543400+21
.text:00543413                 mov     bl, [eax+edx]
.text:00543416                 xor     bl, cl
.text:00543418                 add     cl, 0C1h
.text:0054341B                 mov     [eax+edx], bl
.text:0054341E                 inc     eax
.text:0054341F                 cmp     eax, esi
.text:00543421                 jl      short loc_0_543413
.text:00543423                 pop     ebx
.text:00543424
.text:00543424 loc_0_543424:   ; CODE XREF: sub_0_543400+D
.text:00543424                 pop     esi
.text:00543425                 pop     ebp
.text:00543426                 retn
.text:00543426 sub_0_543400    endp
```

The first section of code up to 00543412 is clearly initialization code, using esi for storing the the position of the end of the data and ebp for the stack pointer (previous values are saved on the stack), setting the initial value (5) of the byte counter cl and checking that the length of the data is not zero – if it is, it jumps to the final section. Lastly, edx is set to the position of the data in memory.

The second section is where the XOR operation takes place. Each byte from the data is moved into bl and XORed in turn with the counter cl, which is subsequently incremented by 0xC1. The XORed byte is then moved back to overwrite the data. A test is made to check if there is more data to process, and if so this section is repeated with the next byte.

The final section restores the saved registers and returns from the function.

Build Tools

This project could not have been built without the following tools, and so an overview of their operation is given in this appendix.

C# Complier Tools – Lexer and Parser Generator

The C# compiler tools (Crowe 2004) come in three main parts – the lexer generator (lg.exe), the parser generator (pg.exe) and the tools library (Tools.dll), which is used by both the generators as well as client applications. There is a reduced-size runtime version of the library which contains only the code necessary to run the generated lexers and parsers – this version is distributed as a part of the MNGTools setup file.

The lexer and parser generators generate code in C# by reading in lexer and parser definition files and converting them into a set of rules. The syntax of the input files is equivalent to that required by the *lex* and *yacc* tools, with some extensions to provide for a better fit with the object-oriented approach of .NET.

More information on the operation and internals of the compiler tools is available on the accompanying CD, or from (Crowe 2004).

NDoc – XML-based Documentation Generator

The development environment produces XML files containing structured comments from the comments in the source files, with assistance from VBCommenter (GotDotNet 2004) in the case of VB.NET. NDoc (2004) takes these XML files and produces easily-navigable Complied HTML help files. The rules for doing so are themselves defined by XML files.

XML comments are straightforward to write, although it is important to remember to terminate all XML elements correctly.

Code Samples

Considering the volume of code required for this project, much of which is not of academic interest, some files have been excluded. All source code is on the accompanying CD and available at the project website. (MNGEdit 2004)

Table F-1: Included code samples

Component	Name	Description
MNGPad	MNGPad.vb	MNG file editor
MNGPlayer	MNGPlayer.vb	MNG music player
MNGTree	ConditionNode.vb	AST node representing a condition
MNGParser	MNGParser.cs	MNG script parser (to AST)
	MNG.lexer	Input file to lexer generator
	MNG.parser	Input file to parser generator
MNGFilter	MNGFile.vb	MNG file container & I/O component
	Sample.vb	Internal sample representation
	SampleHashtable.vb	Hashtable subclass for samples
	ScriptWriter.vb	Visitor subclass used to write out the AST to a text script
	SampleScanner.vb	Visitor subclass generating a list of Waves from the AST (for saving)
MNGEngine	MNGEngine.vb	MNG music playing component
	SoundManager.vb	Sound buffer handler
	VoicePlayer.vb	Manages instances of sample playback
	ExpressionEvaluator.vb	Script expression node evaluator
	LayerReader.vb	Reads layer details from the AST
	LayerPlayer.vb	Controls layer and effect playback
	AleotoricLayerPlayer.vb	Plays voices in Aleotoric layers

Note: Comments prefaced by three apostrophes are intended for use by the XML documenter – those prefaced by just one apostrophe are standard comments.